Concurrent programming: From theory to practice

Concurrent Algorithms 2021

Vasileios Trigonakis
Principal Researcher
Oracle Labs Zurich
Vasileios Trigonakis

- Principal Researcher @ Oracle Labs
- PhD in Computer Science from EPFL
- Started at Oracle in 2016
- Leading the PGX Distributed (PGX.D) project

in/vtrigonakis
### From theory to practice

<table>
<thead>
<tr>
<th>Theoretical (design)</th>
<th>Practical (design)</th>
<th>Practical (implementation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Impossibilities</td>
<td>• System models</td>
<td>• <strong>Hardware</strong></td>
</tr>
<tr>
<td>• Upper/Lower bounds</td>
<td>• shared memory</td>
<td>• Which atomic ops</td>
</tr>
<tr>
<td>• Techniques</td>
<td>• message passing</td>
<td>• Memory consistency</td>
</tr>
<tr>
<td>• System models</td>
<td>• Finite memory</td>
<td>• Cache coherence</td>
</tr>
<tr>
<td>• Correctness proofs</td>
<td>• Practicality issues</td>
<td>• Locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Performance</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Scalability</td>
</tr>
</tbody>
</table>

- **Design (pseudo-code)**
- **Design (pseudo-code, prototype)**
- **Implementation (code)**
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures
Why do we use caching?

Core freq: 2GHz = 0.5 ns / instr
Core → Disk = ~ms
Core → Memory = ~100ns
Cache
- Large = slow
- Medium = medium
- Small = fast
Why do we use caching?

Core freq: 2GHz = 0.5 ns / instr
Core → Disk = ~ms
Core → Memory = ~100ns

Cache
- Core → L3 = ~20ns
- Core → L2 = ~7ns
- Core → L1 = ~1ns
Typical server configurations

**Intel Xeon**
- 14 cores @ 2.4GHz
- L1: 32KB
- L2: 256KB
- L3: 40MB
- Memory: 512GB

**AMD Opteron**
- 18 cores @ 2.4GHz
- L1: 64KB
- L2: 512KB
- L3: 20MB
- Memory: 512GB
Experiment
Throughput of accessing some memory, depending on the memory size
Outline

• CPU caches
• **Cache coherence**
• Placement of data
• Graph processing: Concurrent data structures
Until ~2004: single-cores

- Single core
- Core freq: 3+GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → L2
  - Core → L1
After ~2004: multi-cores

Multiple cores
Core freq: ~2GHz
Core → Disk
Core → Memory
Cache
- Core → shared L3
- Core → L2
- Core → L1

Private

Core 0
L1
L2
L3
Memory
Disk

Core 1
L1
L2

multiple copies
Cache coherence for consistency

Core 0 has X and Core 1
- wants to write on X
- wants to read X
- did Core 0 write or read X?

To perform a write
- invalidate all readers, or
- previous writer
To perform a read
- find the latest copy
Cache coherence with MESI

A state diagram
State (per cache line)
- Modified: the only dirty copy
- Exclusive: the only clean copy
- Shared: a clean copy
- Invalid: useless data

Which state is our “favorite?”
The ultimate goal for scalability

---

A state diagram

State (per cache line)

- **Modified**: the only dirty copy
- **Exclusive**: the only clean copy
- **Shared**: a clean copy
- **Invalid**: useless data

= threads can keep the data close (L1 cache)
= faster
Experiment
The effects of false sharing
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures
Uniformity vs. non-uniformity

Typical desktop machine

Typical server machine

= Uniform

= non-Uniform (aka. NUMA)
Latency (ns) to access data in a NUMA multi-core server

Conclusion: we need to take care of locality
Experiment
The effects of locality
Experiment
The effects of locality

Same memory node

```
vtrigona $ ./test_locality -x0 -y1
Size: 8 counters = 1 cache lines
Thread 0 on core : 0
Thread 1 on core : 1
Number of threads: 2
Throughput : 104.27 Mop/s
```

Different memory nodes

```
vtrigona $ ./test_locality -x0 -y10
Size: 8 counters = 1 cache lines
Thread 0 on core : 0
Thread 1 on core : 10
Number of threads: 2
Throughput : 43.16 Mop/s
```
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures
Your Data is a Graph!

- Represent it as a property graph
  - Entities are vertices
  - Relationships are edges
- Annotate your graph
  - Labels identify vertices and edges
  - Properties describe vertices and edges
- For the purpose of
  - Data modeling
  - Data analysis

Navigate multi-hop relationships quickly (instead of joins)
### Relational (Database) Model → Property Graph Model

#### Users Table
<table>
<thead>
<tr>
<th>user_id (PK)</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vasilis</td>
</tr>
<tr>
<td>1</td>
<td>Lucas</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### User Likes Table
<table>
<thead>
<tr>
<th>user_id</th>
<th>post_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Posts Table
<table>
<thead>
<tr>
<th>author_id</th>
<th>post_id (PK)</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ETH</td>
</tr>
<tr>
<td>123</td>
<td>1</td>
<td>Oracle</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

![Graph Model Diagram]

--property graph model

Essentially having “materialized joins”
Main Approaches of Graph Processing

1. Computational graph analytics [ASPiLOS’12, VLDB’16]
   - Iterate the graph multiple times and compute mathematical properties using Greenmarl / PGX Algorithm (e.g., Pagerank)
   - e.g., `graph.getVertices().forEach(n -> ...)`
2. Graph querying and pattern matching [GRADES’16/17, VLDB’16]
   - Query the graph using PGQL to find sub-graphs that match to the given relationship pattern
   - e.g., `SELECT ... MATCH (a) -[edge]-> (b) ...`
3. Graph ML (new)
   - Use the structural information latent in graphs
   - e.g., graph similarity

\[
PR(p_i) = \frac{1 - d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}
\]
Dissecting a graph processing system
with a focus on (concurrent) data structures
Dissecting a graph processing system and preparing for a job interview with a focus on (concurrent) data structures
Architecture of a graph processing system

Tons of other data and metadata to store
Graph

**tmp graph structure**

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

**graph structure**

user-ids - internal ids

Vasilis → 0
Rachid → 1
Breaking bad → 2
Dexter → 3

Vasilis → 0 → Vasilis
Rachid → 1 → Rachid
Breaking bad → 2 → Breaking bad
Dexter → 3 → Dexter

labels

:likes, :people, :similar, ...

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

Runtime

**indices / metadata**

< 300
> = 300

buffer management

1MB 1MB 1MB 1MB

label by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

task / job scheduling

Producers

Consumers

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

labels

:likes, :people, :similar, :male ...

renaming (ids)

used used used

Operations

**group by / join**

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

distinct

Vasilis
Rachid

Vasilis
Rachid

limit (top k)

BFS

DFS

runtime management

number_of_references: X

used used used
Graph

**tmp graph structure**

- “Vasilis”, “Breaking bad”, :likes
- “Rachid”, “Dexter”, :likes
- “Vasilis”, “Dexter”, :likes
- “Dexter”, “Breaking bad”, :similar
- “Breaking bad”, “Dexter”, :similar

**graph structure**

↓

**user-ids - internal ids**

| Vasilis  | 0 |
| Rachid   | 1 |
| Breaking bad | 2 |
| Dexter   | 3 |

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

**labels**

:likes, :people, :similar, ...

**properties**

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

**lifetime management**

number_of_references: X

**Classic graph structures**

1. **connectivity matrix**

2. **adjacency list**

3. **compressed source row (CSR)**
Graph

**Storing labels**
- usually a small enumeration e.g., person, female, male
- storing strings is expensive
  - “person” → ~ 7 bytes
- comparing strings is expensive
  - dictionary encoding, e.g.,
    - person → 0
    - female → 1
    - male → 2

**Ofc, hash map to**
- store those
- translate during runtime

**user-ids - internal ids**
- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

**hash map / array**
- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

**labels**
- :likes, :people, :similar, …

**properties**
- “Vasilis”, {people, male}, 33, Zurich
- “Rachid”, {people, male}, ??, Lausanne

**lifetime management**
- number_of_references: X
• Property
  • one type per property, e.g., int
  • 1:1 mapping with vertices/edges
    \( \Rightarrow \) (sequential) arrays

• Lifetime management
  (and other counters)
  • cache coherence: atomic counters can be expensive
  • Two potential solutions
  1. approximate counters
  2. stripped counters

Thread local:

```
increment(int by) { counter[my_thread_id] += by; }
int value() {
    int sum = 0;
    for (int i = 0; i < num_threads; i++) { sum += counter[i]; }
    return sum;
}
```
Graph

**tmp graph structure**

```
"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar
```

**graph structure**

```
user-ids - internal ids
Vasilis \(\rightarrow\) 0
Rachid \(\rightarrow\) 1
Breaking bad \(\rightarrow\) 2
Dexter \(\rightarrow\) 3
```

**hash map / array**

```
0 \(\rightarrow\) Vasilis
1 \(\rightarrow\) Rachid
2 \(\rightarrow\) Breaking bad
3 \(\rightarrow\) Dexter
```

**labels**

```
"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ?, Lausanne
```

**properties**

```
```

**lifetime management**

```
number_of_references: X
```

**segmented buffer**

```
```

**CSR**

```
```

**dictionary (= map)**

```
```

**array**

```
```

**stripped counter**

```
```

**Structure**

<table>
<thead>
<tr>
<th>Structure</th>
<th># Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>array / buffer</td>
<td>5</td>
</tr>
<tr>
<td>map</td>
<td>2</td>
</tr>
</tbody>
</table>

---

**Score**

---

Copyright © 2021, Oracle and/or its affiliates. All rights reserved.
• Indices
  • Used for speeding up “queries”
    • Which vertices have label :person?  
    • Which edges have value > 1000? 

→ maps, trees

• Buffer management
  • In “real” systems, resource management is very important
    • buffer pools
      • no order
      • insertions and deletions
      • no keys
  
→ Fixed num object pool: array
→ Otherwise: list
→ Variable-sized elements: heap
- Task and job scheduling
  - produces create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements
→ queues

- Storing / querying sets of labels
  - set equality expensive
  - usually common groups
e.g., \{person, female\}, \{person, male\}
→ 2-level dictionary encoding
  - \{person, female\} → 0
  - \{person, male\} → 1

- Giving unique ids (renaming)
→ tree, map, set, counter, other?
**Runtime**

- **indices / metadata**
  - map / tree
  - less than 300
  - greater than or equal to 300

- **buffer management**
  - array
  - 1MB

- **task / job scheduling**
  - queue
  - task
  - producers
  - consumers

- **labels**
  - :likes, :people, :similar, :male …
  - dictionary (= map)
    - {people, male} → {2,4}

- **renaming (ids)**
  - map / tree / set
  - used

---

**Score**

<table>
<thead>
<tr>
<th>Structure</th>
<th># Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>array / buffer</td>
<td>6</td>
</tr>
<tr>
<td>map</td>
<td>5</td>
</tr>
<tr>
<td>tree / heap</td>
<td>2</td>
</tr>
<tr>
<td>set</td>
<td>1</td>
</tr>
<tr>
<td>queue</td>
<td>1</td>
</tr>
</tbody>
</table>

---

Copyright © 2021, Oracle and/or its affiliates. All rights reserved.
### Operations

#### group by / join
- Vasilis, Breaking bad
- Rachid, Dexter
- Vasilis, Dexter

#### distinct
- Vasilis
- Rachid
- Vasilis

#### limit (top k)

<table>
<thead>
<tr>
<th>11 12 13 12 9 0 8</th>
<th>32 23 3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 9 11 13 32 23</td>
<td>13 23 3 2 0</td>
</tr>
</tbody>
</table>

#### BFS

#### DFS

### Group by
1. Mapping from keys to values
2. Atomic value aggregations
   - e.g., COUNT, SUM, MAX
   - insertion only
   - hash map
   - atomic inc / sum / max, etc.

### Join
- create a map of the small table
- insertion phase, followed by
- probing phase
- hash map, lock-free probing
• Distinct
  • can be solved with sorting, or
- **Distinct**
  - can be solved with sorting, or
  - \( \rightarrow \) **hash set**

- **Limit (top k)**
  - can be solved with sorting, or
  - different specialized structures
  - \( \rightarrow \) **tree**
  - \( \rightarrow \) **heap**
  - \( \rightarrow \) **~ list**
  - \( \rightarrow \) **array** (e.g., 2 elements only)
  - \( \rightarrow \) **register** (1 element only)
• **Breadth-first search (BFS)**
  - FIFO order
  - track visited vertices
  → *queue*
  → *set*

• **Depth-first search (DFS)**
  - LIFO order
  - track visited vertices
  → *stack*
  → *set*
Operations

- group by / join
- map / atomics
- distinct
- hash set
- limit (top k)
- tree / heap / list
- BFS
- queue / set
- DFS
- stack / set

Score

<table>
<thead>
<tr>
<th>Structure</th>
<th># Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>array / buffer</td>
<td>7</td>
</tr>
<tr>
<td>map</td>
<td>6</td>
</tr>
<tr>
<td>set</td>
<td>4</td>
</tr>
<tr>
<td>tree / heap</td>
<td>3</td>
</tr>
<tr>
<td>queue</td>
<td>2</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
</tr>
<tr>
<td>list</td>
<td>1</td>
</tr>
</tbody>
</table>
Your new cheatsheet for interviews!

---

**Graph**

- **Graph structure**
  - CSR
  - Hash map / array
  - Dictionary
  - Array
  - Labels
  - Properties
  - Lifetime management
  - Stripped counter
  - Number of references: X

- **User-ids - Internal ids**
  - Vasilis → 0
  - Rachid → 1
  - Breaking bad → 2
  - Dexter → 3

- **Labels**
  - :likes, :people, :similar, :male

- **Properties**
  - "Vasilis", \{people, male\}, 33, Zurich
  - "Rachid", \{people, male\}, ??, Lausanne

**Runtime**

- **Indices / Metadata**
  - Map / Tree
  - Buffer management
  - Task / Job scheduling
  - Producers
  - Consumers

- **Labels**
  - :likes, :people, :similar, :male

- **Properties**
  - "Vasilis", :likes
  - "Rachid", :likes

- **Map / Tree / Set**
  - Hash set

- **Limit (Top k)**
  - BFS
  - DFS

**Operations**

- **Group by / Join**
  - Map / Atomics

- **Distinct**
  - Hash set

- **Limit (Top k)**
  - BFS
  - DFS

Copyright © 2021, Oracle and/or its affiliates. All rights reserved.
Conclusions

• Both **theory** and **practice** are necessary for
  • Designing, and
  • Implementing fast / scalable data structures
• **Hardware** plays a huge role on implementations
  • How and which memory access patterns to use
• **(Concurrent) Data structures**
  • The backbone of every system
  • An “open” and challenging area or research
Identify, explore, and transfer new technologies that have the potential to substantially improve Oracle's business.

Oracle Labs Mission Statement
Internship and job opportunities

Visit the Oracle Labs Internship Page: labs.oracle.com/pls/apex/labs/r/labs/internships

or find our topics in EPFL’s portal

- Automated Machine Learning with Explainability (AutoMLx)
- Automating OCA Verification of GitHub Pull Requests
- BPF Linux Schedulers
- Extending a Distributed Graph Engine (Oracle Labs PGX)
- Extending a Web-Based Enterprise Data Science Platform
- Graph Machine Learning at Oracle
- Graph Support in the Oracle Database
- Machine Learning and Data Analysis Techniques for Domain Global Graphs
- Machine Learning for Optimizing Oracle Database Performance
- Machine Learning Processing in DB Systems
- Oracle Database Multilingual Engine - Modern Programming Languages in the Database

or just send us an email at epfl-labs_ch@oracle.com
Using the Oracle Cloud for free

Everybody
Oracle Cloud Always-Free Tier: oracle.com/cloud/free/

Universities and Schools
Oracle Academy: academy.oracle.com

Research Institutions
Oracle For Research: oracle.com/oracle-for-research/
Our mission is to help people see data in new ways, discover insights, unlock endless possibilities.