Main Memory Adaptive Indexing for Multi-core Systems

Felix Martin Schuhknecht
Victor Alvarez
Jens Dittrich
Stefan Richter

Information Systems Group
Saarland University
https://infosys.uni-saarland.de/
Problem: Answer Range Queries
Problem: Answer Range Queries

```sql
select A
from R
where R.A >= 10 and R.A < 20
```
Problem: Answer Range Queries

```sql
select A
from R
where R.A >= 10 and R.A < 20
```

One extreme: Scan + Filter

```
R.A (unsorted)

43
9
13
22
19
15
7
99
48
17
34
```

$\geq 10 \land \land < 20 \ ?$
Problem: Answer Range Queries

```sql
select A
from R
where R.A >= 10 and R.A < 20
```

One extreme: Scan + Filter

R.A (unsorted)

<table>
<thead>
<tr>
<th>43</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>99</td>
</tr>
<tr>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>17</td>
<td>34</td>
</tr>
</tbody>
</table>

>= 10 && < 20 ?
Problem: Answer Range Queries

```sql
select A
from R
where R.A >= 10 and R.A < 20
```

One extreme: Scan + Filter

Other extreme: Index

R.A (unsorted)

```
| 17 |
| 19 |
| 22 |
| 43 |
| 48 |
| 99 |
| 7 |
```

R.A (sorted)

```
| 7  |
| 9  |
| 13 |
| 15 |
| 17 |
| 19 |
| 22 |
```

>= 10 && < 20?
Problem: Answer Range Queries

```
select A
from   R
where  R.A >= 10 and R.A < 20
```

One extreme: Scan + Filter

Other extreme: Index

[Diagram showing unsorted and sorted lists]
Index: When to build?

One extreme:  At once (Traditional Indexing)
Index: When to build?

One extreme:  At once (Traditional Indexing)
Index: When to build?

One extreme: At once (Traditional Indexing)
Index: When to build?

One extreme: At once (Traditional Indexing)
Index: When to build?

One extreme: At once (Traditional Indexing)
Index: When to build?

One extreme: **At once (Traditional Indexing)**
Index: When to build?

One extreme: **At once (Traditional Indexing)**
Index: When to build?

One extreme:  At once (Traditional Indexing)
Other extreme: Incrementally at query time (Adaptive Indexing)
Index: When to build?

One extreme:  **At once (Traditional Indexing)**
Other extreme: **Incrementally at query time (Adaptive Indexing)**
Index: When to build?

One extreme:  At once (Traditional Indexing)
Other extreme: Incrementally at query time (Adaptive Indexing)
Index: When to build?

One extreme: At once (Traditional Indexing)

Other extreme: Incrementally at query time (Adaptive Indexing)
Traditional Indexing: Sort + Binary Search

Q1: select * from R where R.A > 10 and R.A < 14

Q2: select * from R where R.A > 7 and R.A <= 16

Index:

A <= 10
10 < A < 14
A >= 14

16 < A

Index:

10 < A < 14
7 < A <= 10
14 <= A <= 16
16 < A
Traditional Indexing: Sort + Binary Search

A

13
16
4
9
2
12
7
1
19
3
14
11
8
6

Index Column (A)

1
2
3
4
6
7
8
9
11
12
13
14
16
19

Sort
Traditional Indexing: Sort + Binary Search

<table>
<thead>
<tr>
<th>A</th>
<th>Index Column (A)</th>
<th>Index Column (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>16</td>
</tr>
</tbody>
</table>

Sort

select A 
from R
where R.A > 10 
and R.A < 14
Traditional Indexing: Sort + Binary Search

- **Q1:** select * from R where R.A > 10 and R.A < 14
- **Q2:** select * from R where R.A > 7 and R.A <= 16

Indexing:
- A <= 10
- 10 < A < 14
- A >= 14

Sort:
- 13 16 4 9 2 12 7 1 19 3 14 11 8 6

Index Column (A):
- 1 2 3 4 6 7 8 9 11 12 13 14 16 19

Select A:
- select A from R where R.A > 10 and R.A < 14

Binary Search:
- 1 2 3 4 6 7 8 9 11 12 13 14 16 19
Traditional Indexing: Sort + Binary Search

<table>
<thead>
<tr>
<th>A</th>
<th>Index Column (A)</th>
<th>Index Column (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>19</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>

Q1: select *
    from R
    where R.A > 10
    and R.A < 14

Q2: select *
    from R
    where R.A > 7
    and R.A <= 16

(a)  
(b)  
(c)
Adaptive Indexing: Standard Cracking

[Database Cracking. S. Idreos, M. Kersten, S. Manegold. In CIDR 2007.]
Adaptive Indexing: Standard Cracking

(a)

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

[Database Cracking. S. Idreos, M. Kersten, S. Manegold. In CIDR 2007.]
Adaptive Indexing: Standard Cracking

```
A
13
16
4
9
2
12
7
1
19
3
14
11
8
6
```

```
select A
from R
where R.A > 10
and R.A < 14
```

[Database Cracking. S. Idreos, M. Kersten, S. Manegold. In CIDR 2007.]
Adaptive Indexing: Standard Cracking

A

| 13 | 16 | 4 | 9 | 2 | 12 | 7 | 1 | 19 | 3 | 14 | 11 | 8 | 6 |

Cracked Column (A)

\[
\text{select } A \\
\text{from } R \\
\text{where } R.A > 10 \\
\text{and } R.A < 14
\]

| 4 | 9 | 2 | 7 | 1 | 3 | 8 | 6 |

[Database Cracking. S. Idreos, M. Kersten, S. Manegold. In CIDR 2007.]
Adaptive Indexing: Standard Cracking

A

13
16
4
9
2
12
7
1
19
3
14
11
8
6

Cracked Column (A)

4
9
2
7
1
3
8
6
13
12
11
16
19
14

select A
from R
where R.A > 10
and R.A < 14

[Database Cracking. S. Idreos, M. Kersten, S. Manegold. In CIDR 2007.]
Adaptive Indexing: Standard Cracking

A

| 13 | 16 | 4  | 9  | 2  | 12 | 7  | 1  | 19 | 3  | 14 | 11 | 8  | 6  |

Cracked Column (A)

| 4  | 9  | 2  | 7  | 1  | 3  | 8  | 6  | 13 | 12 | 11 | 16 | 19 | 14 |

select A
from R
where R.A > 10
and R.A < 14

select A
from R
where R.A > 7
and R.A <= 16

[Database Cracking. S. Idreos, M. Kersten, S. Manegold. In CIDR 2007.]
Adaptive Indexing: Standard Cracking

A

| 13 | 16 | 4 | 9 | 2 | 12 | 7 | 19 | 3 | 14 | 11 | 8 | 6 |

Cracked Column (A)

4  
2  
1  
3  
6  
7  
9  
8  
13 
12 
11 
14 
16 
19

select A  
from R  
where R.A > 10 and R.A < 14

select A  
from R  
where R.A > 7 and R.A <= 16

[Database Cracking. S. Idreos, M. Kersten, S. Manegold. In CIDR 2007.]
Motivation

Standard Cracking
Motivation

Standard Cracking

Stochastic Cracking

Hybrid Cracking

Coarse-granular Index

Sideways Cracking
Motivation

Single-threaded adaptive algorithms

- Standard Cracking
- Stochastic Cracking
- Hybrid Cracking
- Coarse-granular Index
- Sideways Cracking
Motivation

Single-threaded adaptive algorithms

Standard Cracking

Stochastic Cracking

Hybrid Cracking

Coarse-granular Index

Sideways Cracking

Quicksort

Radixsort

Mergesort
Motivation

- Single-threaded adaptive algorithms
  - Standard Cracking
  - Stochastic Cracking
  - Hybrid Cracking
    - Coarse-granular Index
  - Sideways Cracking

- Single-threaded sorting algorithms
  - Quicksort
  - Radixsort
  - Mergesort
Motivation

- Single-threaded adaptive algorithms
  - Standard Cracking
  - Hybrid Cracking
  - Stochastic Cracking
  - Coarse-granular Index
  - Sideways Cracking
- Multi-threaded adaptive algorithms
  - Parallel Standard Cracking

- Single-threaded sorting algorithms
  - Quicksort
  - Radixsort
  - Mergesort
Motivation

- Single-threaded adaptive algorithms
  - Standard Cracking
  - Stochastic Cracking
  - Hybrid Cracking
  - Coarse-granular Index
  - Sideways Cracking

- Single-threaded sorting algorithms
  - Quicksort
  - Radixsort
  - Mergesort

- Multi-threaded adaptive algorithms
  - Parallel Standard Cracking
  - ?
Setup
Setup

100 million entries
Setup

100 million entries

Key RowID

100 million entries
Setup

100 million entries

4 Byte + 4 Byte

Key
RowID

100 million entries
Setup

4 Byte + 4 Byte

Key

RowID

Uniform Random Key Distribution

100 million entries

100 million entries
Setup

4 Byte + 4 Byte

Key | RowID

Uniform Random Key Distribution

100 million entries

Query 1%
Setup

100 million entries

Key

RowID

4 Byte + 4 Byte

Uniform Random Key Distribution

Query 1%

~762 MB

100 million entries
Setup

24 GB RAM

Xeon E5-2407

2.2 GHz

2.2 GHz

10MB L3

Xeon E5-2407

2.2 GHz

2.2 GHz

2.2 GHz

2.2 GHz

10MB L3

24 GB RAM
Setup

24 GB RAM

Xeon E5-2407

2.2 GHz 2.2 GHz

10MB L3

Xeon E5-2407

2.2 GHz 2.2 GHz

2.2 GHz 2.2 GHz

10MB L3

24 GB RAM

No Turbo, no HyperThreading
Single-threaded algorithms

Accumulated Query Response Time [s]

1 Thread, 100 Million Elements

- Standard Cracking (SC)
- Hybrid Crack Sort (HCS)
- Coarse-granular Index (CGI)
- Radix Sort (RS)
- STL std::sort (STL-S)

[The Uncracked Pieces in Database Cracking. F. M. Schuhknecht, A. Jindal, J. Dittrich. In PVLDB 2013]
Single-threaded algorithms

Accumulated Query Response Time [s]

Query Sequence

1 Thread, 100 Million Elements

- Standard Cracking (SC)
- Hybrid Crack Sort (HCS)
- Coarse-granular Index (CGI)
- Radix Sort (RS)
- STL std::sort (STL-S)

750 Queries

[The Uncracked Pieces in Database Cracking. F. M. Schuhknecht, A. Jindal, J. Dittrich. In PVLDB 2013]
Single-threaded algorithms

1 Thread, 100 Million Elements

- Standard Cracking (SC)
- Hybrid Crack Sort (HCS)
- Coarse-granular Index (CGI)
- Radix Sort (RS)
- STL std::sort (STL-S)

Accumulated Query Response Time [s]

Query Sequence

[The Uncracked Pieces in Database Cracking. F. M. Schuhknecht, A. Jindal, J. Dittrich. In PVLDB 2013]
Multi-threaded environments?

Multi-threaded algorithms!
Multi-threaded algorithms:
Parallel Standard Cracking (P-SC)

Multi-threaded algorithms:
Parallel Standard Cracking (P-SC)

Multi-threaded algorithms:
Parallel Standard Cracking (P-SC)

Multi-threaded algorithms: Parallel Standard Cracking (P-SC)

Multi-threaded algorithms: Parallel Standard Cracking (P-SC)

Requested Locks

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>T2</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W</td>
</tr>
</tbody>
</table>

Multi-threaded algorithms: Parallel Standard Cracking (P-SC)

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>W</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>T2</td>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Multi-threaded algorithms: Parallel Standard Cracking (P-SC)

Inter-query parallelism

Requested Locks

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</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>W</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>W</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>W</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Multi-threaded algorithms: Parallel Standard Cracking (P-SC)

Multi-threaded algorithms: Parallel Standard Cracking (P-SC)

Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)

Query
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)

Query

k Chunks
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms: Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)

Query

Cracker Index

T1

Cracker Index

T2

Cracker Index

T3

Cracker Index

Tk

k Chunks

Fully utilize resources

Complete independence
Multi-threaded algorithms:
Parallel-chunked Standard Cracking (P-CSC)

Query

T1
Cracker
Index

T2
Cracker
Index

T3
Cracker
Index

Tk
Cracker
Index

k Chunks

Fully utilize resources

Complete independence

No consecutive result
Micro Benchmark
Reading 1% from k locations using one thread
Micro Benchmark
Reading 1% from k locations using one thread

No problem for realistic k
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI)
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying

A

Index(A)

1024 partitions
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying

Index on A

A

Index(A)

1024 partitions

Cracker Index
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying
2. Perform P-SC
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying

2. Perform P-SC

1024 partitions
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying

2. Perform P-SC Index(A)

Index(A)

1024 partitions

W
R
R
W

Like starting ...

Cracker Index

... after 1000 cracks
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying
   - Index on A
   - Index on A

2. Perform P-SC
   - Query
   - W
   - R
   - R
   - W

1024 partitions

Like starting ...

... after 1000 cracks

Reduces lock contention of P-SC
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying

2. Perform P-SC

Index on A

1024 partitions

Reduces lock contention of P-SC

Like starting ... after 1000 cracks

Reduces Variance

W

R

R

W

Cracker Index

A

Query
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying
   - Index on A

2. Perform P-SC
   - Adds (small) initialization time
   - Reduces lock contention of P-SC
   - Reduces Variance

Like starting ...
... after 1000 cracks

1024 partitions

Cracker Index

W
R
R
W
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI)

1. Range-partition while copying Index on A

2. Perform P-SC Index(A)

- Adds (small) initialization time
- 1024 partitions
- Reduces lock contention of P-SC

Query

W
R
R
W

1024 partitions

Like starting ...

... after 1000 cracks

Reduces Variance
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI):
Parallel Range Partitioning

# Elements
# Threads (k)

Source

Thread 1
Thread 2
...
Thread k

Destination
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI):
Parallel Range Partitioning

1. Build Histogram

16 / 30
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI): Parallel Range Partitioning

1. Build Histogram

Source

Destination

Range-partition

Thread 1
Thread 2

Thread k

16 / 30
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI):
Parallel Range Partitioning

1. Build Histogram
2. Copy entries

Source

Destination

Range-partition

# Elements

# Threads (k)

Thread 1

Thread 2

Thread k

Thread 1

Thread 2

Thread k
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI):
Parallel Range Partitioning

1. Build Histogram
2. Copy entries
Multi-threaded algorithms:
Parallel Coarse-Granular Index (P-CGI):
Parallel Range Partitioning

1. Build Histogram
2. Copy entries

Source

| # Elements |
|-----|---|
| # Threads (k) |

Thread 1
Thread 2
... 
Thread k

Destination

Range-partition

No locks required

Thread 1
Thread 2
... 
Thread k
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI): Parallel Range Partitioning

1. Build Histogram
2. Copy entries

Source

<table>
<thead>
<tr>
<th># Elements</th>
<th># Threads (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td></td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Thread k</td>
<td></td>
</tr>
</tbody>
</table>

Destination

Range-partition

Thread 1
Thread 2
... Thread k

No locks required

Fully utilize resources
Multi-threaded algorithms: Parallel Coarse-Granular Index (P-CGI): Parallel Range Partitioning

1. Build Histogram
2. Copy entries

# Elements

\[ \frac{\text{# Elements}}{\text{# Threads (k)}} \]

1. Build Histogram
2. Copy entries

 Fully utilize resources

Range-partition

Thread 1
Thread 2
...
Thread k

Source

Destination

NUMA-fragmented memory

No locks required
Multi-threaded algorithms: Parallel-chunked Standard Cracking (P-CSC)
Multi-threaded algorithms:
Parallel-chunked Coarse-Granular Index (P-CCGI)
Multi-threaded algorithms:
Parallel-chunked Coarse-Granular Index (P-CCGI)

Range-partitioning

k Chunks
Multi-threaded algorithms: Parallel-chunked Coarse-Granular Index (P-CCGI)
Multi-threaded algorithms:
Parallel-chunked Coarse-Granular Index (P-CCGI)
Multi-threaded algorithms:
Parallel-chunked Coarse-Granular Index (P-CCGI)

P-CSC + Range Partitioning

Query

Range-partitioning

Cracker Index

Cracker Index

Cracker Index

Cracker Index

T1

Local Result

T2

Local Result

T3

Local Result

Tk

Local Result

k Chunks
Multi-threaded algorithms:
Parallel Range-Partitioned Radix Sort (P-RPRS)
Multi-threaded algorithms:
Parallel Range-Partitioned Radix Sort (P-RPRS)

1. Range-partition while copying

1024 partitions

A

Index(A)
Multi-threaded algorithms:
Parallel Range-Partitioned Radix Sort (P-RPRS)

1. Range-partition while copying

2. Perform in-place radix sort on each partition
Multi-threaded algorithms: Parallel Range-Partitioned Radix Sort (P-RPRRS)

1. Range-partition while copying

2. Perform in-place radix sort on each partition

1024 partitions

Fully sorted
Multi-threaded algorithms: Parallel Range-Partitioned Radix Sort (P-RPRS)

1. Range-partition while copying

2. Perform in-place radix sort on each partition

- 1024 partitions
- 256 bucket
- Most significant byte → 4 recursion levels

Index(A)

A

Index(A)

Index(A)

Fully sorted

shared with P-CCGI
Multi-threaded algorithms:
Parallel Range-Partitioned Radix Sort (P-RPRS)

**Graph:**
- **P-RPRS**
- (Parallel) Mergesort (GNU libstdc++)

**Data:**
- 4 Cores / 8 Threads
- 512 million 4 byte integers
- Uniform random distribution
Multi-threaded algorithms:
Parallel-chunked Range-Partitioned Radix Sort (P-CRS)

Range-partitioning

```
<table>
<thead>
<tr>
<th>T1</th>
<th>CGI + RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>CGI + RS</td>
</tr>
<tr>
<td>T3</td>
<td>CGI + RS</td>
</tr>
<tr>
<td>Tk</td>
<td>CGI + RS</td>
</tr>
</tbody>
</table>
```

Query

\( k \) Chunks
Multi-threaded algorithms:
Parallel-chunked Range-Partitioned Radix Sort (P-CRS)
Multi-threaded algorithms:
Parallel-chunked Range-Partitioned Radix Sort (P-CRS)

P-RPRS + Chunking

Range-partitioning

Query

T1
CGI + RS

Local Result

T2
CGI + RS

Local Result

T3
CGI + RS

Local Result

Tk
CGI + RS

Local Result

k Chunks
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

Query Sequence

P-SC
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

P-SC
P-CSC (Chunks * SC)

Query Sequence
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

P-SC
P-CSC
P-CGI
P-CCGI
P-RPRS
P-CRS

lock vs. lock-free

(#Chunks * SC)
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

P-SC
P-CSC --- (#Chunks * SC)

lock vs. lock-free

almost 2x faster

Query Sequence

21 / 30
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s] vs. Query Sequence

- **P-SC**
- **P-CSC**
- **P-CGI**
- **P-CCGI**
- **P-RPRS**
- **P-CRS**

(Chunks * SC)
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

- P-SC
- P-CSC (\#Chunks * SC)
- P-CGI (Par. Range Partitioning + P-SC)

Query Sequence

21 / 30
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

P-SC
P-CSC (\#Chunks * SC)
P-CGI (Par. Range Partitioning + P-SC)

faster despite of range partitioning
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

Query Sequence

P-SC
P-CSC: (#Chunks * SC)
P-CGI: (Par. Range Partitioning + P-SC)
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

- P-SC
- P-CSC: (#Chunks * SC)
- P-CGI: (Par. Range Partitioning + P-SC)
- P-CCGI: (#Chunks * (Range Partitioning + SC))
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

Query Sequence

- P-SC
- P-CSC  
  (#Chunks * SC)
- P-CGI  
  (Par. Range Partitioning + P-SC)
- P-CCGI  
  (#Chunks * (Range Partitioning + SC))

huge improvement in querying
Multi-threaded Results

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

Query Sequence
Multi-threaded Results

8 Threads, 100 Million Elements

- P-SC
- P-CSC : (#Chunks * SC)
- P-CGI : (Par. Range Partitioning + P-SC)
- P-CCGI : (#Chunks * (Range Partitioning + SC))
- P-RPRS : (Par. Range Partitioning + #Parts * RS)
Multi-threaded Results

8 Threads, 100 Million Elements

- **P-SC**: (#Chunks * SC)
- **P-CSC**: (#Chunks * (Range Partitioning + SC))
- **P-CGI**: (Par. Range Partitioning + P-SC)
- **P-CCGI**: (Par. Range Partitioning + #Parts * RS)
- **P-RPRS**: (Par. Range Partitioning + #Parts * RS)

Accumulated Query Response Time [s] vs. Query Sequence

*most expensive initialization*
Multi-threaded Results

8 Threads, 100 Million Elements

- **P-SC**: (#Chunks * SC)
- **P-CSC**: (Par. Range Partitioning + P-SC)
- **P-CGI**: (#Chunks * (Range Partitioning + SC))
- **P-CCGI**: (Par. Range Partitioning + #Parts * RS)
- **P-RPRS**: (Par. Range Partitioning + #Parts * RS)
Multi-threaded Results

8 Threads, 100 Million Elements

- **P-SC**
- **P-CSC** (Chunks * SC)
- **P-CGI** (Parallel Range Partitioning + P-SC)
- **P-CCGI** (Chunks * (Range Partitioning + SC))
- **P-RPRS** (Parallel Range Partitioning + Parts * RS)
- **P-CRS** (Chunks * (Range Partitioning + RS))
Multi-threaded Results

8 Threads, 100 Million Elements

- **P-SC**
- **P-CSC**
- **P-CGI**
- **P-CCGI**
- **P-RPRS**
- **P-CRS**

Accumulated Query Response Time [s]

- **P-SC**
- **P-CSC**
- **P-CGI**
- **P-CCGI**
- **P-RPRS**
- **P-CRS**

(chunked)

(Par. Range Partitioning + P-SC)

(Par. Range Partitioning + P-SC)

(Par. Range Partitioning + #Parts * RS)

(Par. Range Partitioning + RS)

#Chunks * (Range Partitioning + SC))

#Chunks * (Range Partitioning + RS)
Multi-threaded Results
Factor Speedup from 1 to 8 Threads

- Initialization (including first query)
- Total

Speedup [times]

P-SC
P-SC: Analysis

Bandwidth

Lock Time

<table>
<thead>
<tr>
<th>Mutex</th>
<th>Wait Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piece lock</td>
<td>11.671</td>
</tr>
<tr>
<td>Cracker index lock</td>
<td>5.169</td>
</tr>
<tr>
<td>Total</td>
<td>16.84</td>
</tr>
<tr>
<td>Average (Total by 8)</td>
<td>2.105</td>
</tr>
</tbody>
</table>

Intel VTune Amplifier XE 2013 Data
P-SC: Analysis

Bandwidth

```
<table>
<thead>
<tr>
<th>Mutex</th>
<th>Wait Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piece lock</td>
<td>11.671</td>
</tr>
<tr>
<td>Cracker index lock</td>
<td>5.169</td>
</tr>
<tr>
<td>Total</td>
<td>16.84</td>
</tr>
<tr>
<td>Average (Total by 8)</td>
<td>2.105</td>
</tr>
</tbody>
</table>
```

Intel VTune Amplifier XE 2013 Data
Multi-threaded Results
Factor Speedup from 1 to 8 Threads

- Initialization (including first query)
- Total

<table>
<thead>
<tr>
<th>Method</th>
<th>Speedup [times]</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-SC</td>
<td>1.24</td>
</tr>
<tr>
<td>P-CGI</td>
<td>6.30</td>
</tr>
<tr>
<td>P-RPRS</td>
<td>8.00</td>
</tr>
</tbody>
</table>

24 / 30
Non-Chunked Algorithms: Analysis (P-RPRS)

Range Partitioning (RP) Phase:

![Diagram showing range partitioning process]
Non-Chunked Algorithms: Analysis (P-RPRS)

Range Partitioning (RP) Phase:

Range Partitioning (RP) Phase:

Source

<table>
<thead>
<tr>
<th># Elements</th>
<th># Threads (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Socket 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread k</td>
<td>Socket 2</td>
</tr>
</tbody>
</table>

Destination

| Thread 1   |
| Thread 2   |
|            |
| Thread k   |

25 / 30
Non-Chunked Algorithms: Analysis (P-RPRS)

Range Partitioning (RP) Phase:

![Diagram showing the Range Partitioning (RP) Phase with Source and Destination blocks connected by threads. The diagram illustrates the distribution of elements among threads and sockets.]
Non-Chunked Algorithms: Analysis (P-RPRS)

Range Partitioning (RP) Phase:

Source

# Elements

# Threads (k)

Destination

Thread 1
Socket 1

Thread 2

::

Thread k
Socket 2

Socket 1

Thread 1

Thread 2

::

Thread k

NUMA 1

Thread 1

Thread 2

::

Thread k

NUMA 2

Socket 2

NUMA 1

Thread 1

Thread 2

::

Thread k
Non-Chunked Algorithms: Analysis (P-RPRS)

Query Phase:

Destination

Socket 1
Socket 2

NUMA 1
NUMA 2

Thread 1
Thread 2
Thread k
Non-Chunked Algorithms: Analysis (P-RPRS)

Query Phase:
Non-Chunked Algorithms: Analysis (P-RPRS)

Query Phase:
Multi-threaded Results
Factor Speedup from 1 to 8 Threads
Chunked Algorithms: Analysis (P-CRS)

All chunks are completely independent - 8x Speedup?
Chunked Algorithms: Analysis (P-CRS)

All chunks are completely independent - 8x Speedup?
Chunked Algorithms: Analysis (P-CRS)

All chunks are completely independent - 8x Speedup?
Chunked Algorithms: Analysis (P-CRS)

All chunks are completely independent - 8x Speedup?

2.9x Speedup

Core

Core

Core

Core

Chunk

Chunk

Chunk

Chunk

Shared LLC

2.9x Speedup

Core

Core

Core

Core

Chunk

Chunk

Chunk

Chunk

Shared LLC
Chunked Algorithms: Analysis (P-CRS)

All chunks are completely independent - 8x Speedup?

2.9x Speedup

Chunk

Core

Shared LLC

Main Memory (NUMA Region 1)

2.9x Speedup

Chunk

Core

Main Memory (NUMA Region 2)
Chunked Algorithms: Analysis (P-CRS)

All chunks are completely independent - 8x Speedup?
Conclusion
Conclusion

Initialization Time [s]

Number of Threads

RS    P-RPRS    P-CRS
Conclusion

- RS
- P-RPRS
- P-CRS

Initialization Time [s]

Number of Threads

100 million in less than a second
Conclusion

Initialization Time [s]

Number of Threads

100 million in less than a second

Total Time [s]

non-chunked chunked
Conclusion

- RS
- P-RPRS
- P-CRS

**Initialization Time [s]**

- **Number of Threads**
- **100 million in less than a second**

**Total Time [s]**
- **non-chunked**
- **chunked**

8 Threads, 100 Million Elements

- Accumulated Query Response Time [s]
  - P-CCGi
  - P-CRS

8 Threads, 100 Million Elements

- **Query Sequence**

- **Accumulated Query Response Time [s]**
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8

- **Accumulated Query Response Time [s]**
  - 0
  - 0.5
  - 1
  - 1.5
  - 2
  - 2.5
  - 3
Conclusion

100 million in less than a second

> 10000 queries to win over best cracking
Conclusion

- RS
- P-RPRS
- P-CRS

 Initialization Time [s]

- 100 million in less than a second

- Number of Threads

- Total Time [s]

- > 10000 queries to win over best cracking

- Gap decreased from 5 seconds (1T) to 0.5 seconds (8T)
Upcoming
Upcoming

8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

Query Sequence

P-CCGI

P-CRS
Upcoming

8 Threads, 100 Million Elements

Uses plain Standard Cracking inside
8 Threads, 100 Million Elements

Accumulated Query Response Time [s]

Query Sequence

P-CCGI
P-CRS

Uses plain Standard Cracking inside

Improvable? Next talk!
Thank you!