I/O- and Cache-Efficient Algorithms

Mark de Berg

TU Eindhoven
• a simple and efficient algorithm for smallest enclosing circle . . .
• . . . and its experimental analysis
• cache-oblivious algorithms
  – the model
  – smallest enclosing circle
  – more examples
• conclusions
Compute smallest enclosing circle of set $P$ of $N$ points in the plane.
Compute smallest enclosing circle of set \( P \) of \( N \) points in the plane.
SmallestCircle($P$)
1. RandomPermute($P$)
2. $D :=$ smallest circle for $P[1], P[2], P[3]$
3. for $i := 4$ to $N$
4.   do if $P[i] \in D$
5.      then skip
6.   else $D :=$ smallest circle for $\{P[1], \ldots, P[i]\}$
   where $P[i]$ is on the boundary
SmallestCircle($P$)
1. RandomPermute($P$)
2. $D :=$ smallest circle for $P[1], P[2], P[3]$
3. for $i := 4$ to $N$
4. do if $P[i] \in D$
5. then skip
6. else $D :=$ smallest circle for \{$P[1], \ldots, P[i]$\}
   where $P[i]$ is on the boundary
SmallestCircle($P$)

1. \textit{RandomPermute($P$)}

2. $D :=$ smallest circle for $P[1], P[2], P[3]$

3. \textbf{for} $i := 4$ \textbf{to} $N$

4. \textbf{do if} $P[i] \in D$

5. \hspace{1cm} \textbf{then skip}

6. \textbf{else} $D :=$ smallest circle for \{$P[1], \ldots, P[i]$\}
   \hspace{1cm} where $P[i]$ is on the boundary

"recursive" call
SmallestCircle\((P)\)
1. RandomPermute\((P)\)
2. \(D := \text{smallest circle for } P[1], P[2], P[3]\)
3. \(\text{for } i := 4 \text{ to } N\)
4. \(\text{do if } P[i] \in D\)
5. \(\text{then skip}\)
6. \(\text{else } D := \text{smallest circle for } \{P[1], \ldots, P[i]\}\)
   \(\text{where } P[i] \text{ is on the boundary}\)

RandomPermute\((P)\)
1. \(\text{for } i := 1 \text{ to } N - 1\)
2. \(\text{do } r := \text{random integer in range } i \ldots N\)
3. \(\text{swap } P[i] \text{ and } P[r]\)
Smallest enclosing circle: analysis (1)

\[ \text{RandomPermute}(P) \]
1. \textbf{for} \; i := 1 \textbf{ to } N - 1
2. \textbf{do} \; r := \text{random integer in range } i \ldots N
3. \text{swap } P[i] \text{ and } P[r]

\text{running time is } O(N)
SmallestCircle($P$)
1. RandomPermute($P$)
2. $D :=$ smallest circle for $P[1], P[2], P[3]$
3. for $i := 4$ to $N$
4. do if $P[i] \in D$
5. then skip
6. else $D :=$ smallest circle for $\{P[1], \ldots, P[i]\}$

where $P[i]$ is on the boundary

E[running time]

- $P[1]$ t/m $P[i]$
- $P[i + 1]$ t/m $P[N]$
SmallestCircle($P$)
1. $\text{RandomPermute}(P)$
2. $D := \text{smallest circle for } P[1], P[2], P[3]$
3. for $i := 4$ to $N$
4. do if $P[i] \in D$
5. then skip
6. else $D := \text{smallest circle for } \{P[1], \ldots, P[i]\}$
   where $P[i]$ is on the boundary

$E[\text{running time}]$
$= O(n) + \sum_{i=4}^{N} E[\text{time for } i\text{-th iteration}]$
SmallestCircle($P$)

1. RandomPermute($P$)
2. $D := \text{smallest circle for } P[1], P[2], P[3]$
3. for $i := 4$ to $N$
4. do if $P[i] \in D$
5. then skip
6. else $D := \text{smallest circle for } \{P[1], \ldots, P[i]\}$
   where $P[i]$ is on the boundary

$E[\text{running time}]$

$= O(n) + \sum_{i=4}^{N} E[\text{time for } i\text{-th iteration}]$

$= O(n) + \sum_{i=4}^{N} (O(1) + Pr[P[i] \notin D] \cdot O(i))$
SmallestCircle($P$)
1. RandomPermute($P$)
2. $D := \text{smallest circle for } P[1], P[2], P[3]$
3. for $i := 4$ to $N$
4. do if $P[i] \in D$
5. then skip
6. else $D := \text{smallest circle for } \{P[1], \ldots, P[i]\}$
   where $P[i]$ is on the boundary

$E[\text{running time}]$

= $O(n) + \sum_{i=4}^{N} E[\text{time for } i\text{-th iteration}]$

= $O(n) + \sum_{i=4}^{N} (O(1) + \Pr[P[i] \notin D] \cdot O(i))$

$\leq O(n) + \sum_{i=4}^{N} (O(1) + 3/i \cdot O(i))$
SmallestCircle\( (P) \)
1. \texttt{RandomPermute}(P)
2. \( D := \text{smallest circle for } P[1], P[2], P[3] \)
3. \textbf{for } \( i := 4 \) \textbf{to } N \textbf{ do if } P[i] \in D \textbf{ then skip }
4. \textbf{else } \( D := \text{smallest circle for } \{P[1], \ldots, P[i]\} \)
5. \textbf{where } P[i] \text{ is on the boundary }

\[ E[\text{running time}] = O(n) + \sum_{i=4}^{N} E[\text{time for } i\text{-th iteration}] \]
\[ = O(n) + \sum_{i=4}^{N} (O(1) + \Pr[P[i] \not\in D] \cdot O(i)) \]
\[ \leq O(n) + \sum_{i=4}^{N} (O(1) + 3/i \cdot O(i)) \]
\[ = O(n) \]
Pentium 4, 2.60GHz
≈ 89 MB main memory available to the program
Analysis of algorithms: massive data sets

\[ T(n) = \text{# elementary operations} \] the algorithm performs in the worst case as function of \( N \), the number of input elements

additions, multiplications, comparisons, reading a value from memory, etc.

Hmmm . . . is this justified?
The analysis of algorithms: massive data sets

\[ T(n) = \# \text{elementary operations} \quad \text{the algorithm performs in the worst case as function of } N, \text{the number of input elements} \]

additions, multiplications, comparisons, reading a value from memory, etc.

Hmmm . . . is this justified?

NO!

operations on data in main memory: tens of nanoseconds
disk operations: several milliseconds
Smallest enclosing circle: experiments (2)

Pentium 4, 2.60GHz
≈ 89 MB main memory available to the program
I/O-efficient algorithms: the model (Aggarwal, Vitter '88)

$M =$ size of main memory

$B =$ block size for data transport

"typical value" 8KB
I/O-efficient algorithms: the model (Aggarwal, Vitter '88)

- let algorithm handle data placement and transport
  - which data are placed together in a block
  - which blocks are kept in main memory
- analyze number of disk operations (in terms of $N$, $B$, and $M$)

$M =$ size of main memory

$B =$ block size for data transport

"typical value" 8KB
RandomPermute\((P)\)
1. for \(i := 1\) to \(N - 1\) 
2. do \(r := \text{random integer in range } i \ldots N\) 
3. swap \(P[i]\) and \(P[r]\)

analysis of (expected) number of disk operations

- \(N \leq M\): 0
  
  0 disk operations

- \(N > M\):
  
  \((N - 1) \cdot (1 - \frac{M}{N})\) disk operations
  
  (e.g. \((N - 1)/2\) disk operations when \(N = 2M\))
- $M$ and $B$ depend on platform
- even on fixed machine values of $M$ and $B$ may vary
  - main memory may have to be shared with other processes
  - disk-cache ”changes” block size
- two-level I/O-model too simplistic
Intel Itanium2 memory hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Capacity</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU registers</td>
<td>16 KB</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L1 cache</td>
<td>256 KB</td>
<td>5+ cycles</td>
</tr>
<tr>
<td>L2 cache</td>
<td>6MB</td>
<td>12+ cycles</td>
</tr>
<tr>
<td>L3 cache</td>
<td>2 GB</td>
<td>&gt; 150 cycles</td>
</tr>
<tr>
<td>Main memory</td>
<td></td>
<td>can be $10^6$ cycles</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion: caching behavior can also make a large difference.
Ideal: algorithm that is efficient w.r.t. disk and all cache levels

- caches are not under control of algorithm
- algorithms taking all cache-levels into account quite complicated

So what can we do ??
Algorithm designed for simple two-level memory model

Algorithm is not allowed to use the value of $B$ and $M$!
Cache-oblivious algorithms: assumptions (1)

Assumptions:

- $M$ = size of fast memory
- $B$ = block size for data transport
Cache-oblSpacer oblivious algorithms: assumptions (1)

Assumptions:

- Blocks formed following order in which data is written to ”disk”
Cache-oblivious algorithms: assumptions (1)

Assumptions:

- Blocks formed following order in which data is written to "disk"

\[ M = \text{size of fast memory} \]

\[ B = \text{block size for data transport} \]
Assumptions:

- Blocks formed following order in which data is written to "disk"
Assumptions:

- Blocks formed following order in which data is written to "disk"
• Blocks formed following order in which data is written to "disk"

data
layout on disk
or
or . . .
Cache-oblivious algorithms: assumptions (2)

- Blocks formed following order in which data is written to "disk"

  \[
  \begin{array}{c}
  \text{data} \\
  \text{layout on disk}
  \end{array}
  \]

- Operating system uses optimal replacement strategy
  Note: number of cache misses for LRU is within constant from optimal [...]

- Cache is fully associative

- Tall cache assumption: \( M = \Omega(B^2) \)
Cache-oblivious algorithms: assumptions (2)

- Blocks formed following order in which data is written to "disk"

\[
\text{data} \quad \text{layout on disk} \quad \text{or} \quad \text{or} \quad \text{or} \ldots
\]

- Operating system uses optimal replacement strategy
  Note: number of cache misses for LRU is within constant from optimal [...]

- Cache is fully associative

- Tall cache assumption: \( M = \Omega(B^2) \)

Then: Cache-oblivious algorithm that is efficient in the 2-level memory model is efficient with respect to all cache levels, disk, etc!
Example 1: Smallest enclosing disk
A cache-oblivious algorithm for smallest enclosing circle

\textbf{CacheObliviousSmallestCircle}(P)

1. \textbf{if} (\# points in \(P\)) \(\leq 3\)
2. \textbf{then} return smallest circle for \(P\)
3. \textbf{else} \((P_1, P_2) := \text{RandomSplit}(P)\)
4. \hspace{1em} \(D := \text{CacheObliviousSmallestCircle}(P_1)\)
5. \hspace{1em} \textbf{for all} \(P[i] \in P_2\)
6. \hspace{2em} \textbf{do if} \(P[i] \notin D\)
7. \hspace{3em} \textbf{then} \(D_i := \text{smallest circle for } P \text{ with } P[i] \text{ on its boundary}\)
8. \hspace{1em} \textbf{return} best of all computed circles

With: S. Cabello, X. Goaoc, M. Schroders
A cache-oblivious algorithm for smallest enclosing circle

CacheObliviousSmallestCircle\((P)\)
1. if \((\# \text{ points in } P) \leq 3\)
2. then return smallest circle for \(P\)
3. else \((P_1, P_2) := \text{RandomSplit}(P)\)
4. \(D := \text{CacheObliviousSmallestCircle}(P_1)\)
5. for all \(P[i] \in P_2\)
6. do if \(P[i] \notin D\)
7. then \(D_i := \text{smallest circle for } P \text{ with } P[i] \text{ on its boundary}\)
8. return best of all computed circles

RandomSplit\((P)\)
1. for \(i := 1 \text{ to } N\)
2. do \(r := \text{random number in range} [0, 1]\)
3. if \(r < 1/2\)
4. then Put \(P[i]\) into \(P_1\)
5. else Put \(P[i]\) into \(P_2\)
6. return \((P_1, P_2)\)

With: S. Cabello, X. Goaoc, M. Schroders
old algorithm:

\[
\text{RandomPermute}(P)
\]

1. for \( i := 1 \) to \( N - 1 \)
2. do \( r := \) random integer in range \( i \ldots N \)
3. swap \( P[i] \) and \( P[r] \)

\[
\mathbb{E}[\#\text{cache misses}] = (N - 1) \cdot \left(1 - \frac{M}{N}\right)
\]
old algorithm:

\[ \text{RandomPermute}(P) \]
1. \textbf{for} \( i := 1 \) \textbf{to} \( N - 1 \)
2. \textbf{do} \( r := \text{random integer in range } i \ldots N \)
3. \textbf{swap} \( P[i] \) \textbf{and} \( P[r] \)

\[ \mathbb{E}[\#\text{cache misses}] = (N - 1) \cdot \left(1 - \frac{M}{N}\right) \]

new algorithm:

\[ \text{RandomSplit}(P) \]
1. \textbf{for} \( i := 1 \) \textbf{to} \( N \)
2. \textbf{do} \( r := \text{random number in range } [0, 1] \)
3. \textbf{if} \( r < 1/2 \) \textbf{then} Put \( P[i] \) into \( P_1 \) \textbf{else} Put \( P[i] \) into \( P_2 \)
4. \textbf{return} \( (P_1, P_2) \)

layout on disk: 

\[ \mathbb{E}[\#\text{cache misses}] = \text{Scan}(N) \leq 1 + N/B \]
Smallest enclosing disk: experiments

User (time) and system (stime) time for points on a line.

Pentium 4, 2.60GHz
≈ 89 MB main memory available to the program
Example II: Search trees
binary search tree: search structure for internal memory

- nodes contain one key, have degree 2
- depth is $O(\log_2 N)$
binary search tree: search structure for internal memory

- nodes contain one key, have degree 2
- depth is $O(\log_2 N)$

B-tree: I/O-efficient variant

- nodes contain many keys, have high degree
- put each node into one block on disk: degree is $\Theta(B)$
- depth is $O(\log_B N)$
binary search tree: search structure for internal memory

- nodes contain one key, have degree 2
- depth is $O(\log_2 N)$

B-tree: I/O-efficient variant

- nodes contain many keys, have high degree
- put each node into one block on disk: degree is $\Theta(B)$
- depth is $O(\log_B N)$

in practice, degree is 250 – 2000 and depth is at most 4
regular (cache-aware) B-tree:
• blocks: subtrees of size $B$

search visits $O(\log_B N)$ blocks
regular (cache-aware) B-tree:
• blocks: subtrees of size $B$

search visits $O(\log_B N)$ blocks

cache-oblivious B-tree: (VEB-layout)
• cut tree into subtree at middle level; gives 1 top tree, $\sqrt{N}$ lower trees
• first, write top to disk recursively
• next, write lower trees to disk recursively
cache-oblivious B-tree: (VEB-layout)

- cut tree into subtree at middle level; gives 1 top tree, $\sqrt{N}$ lower trees
- first, write top to disk recursively
- next, write lower trees to disk recursively

Theorem: Number of cache misses for a search is $O(\log_B N)$. 
cache-oblivious B-tree: (VEB-layout)

- cut tree into subtree at middle level; gives 1 top tree, $\sqrt{N}$ lower trees
- first, write top to disk recursively
- next, write lower trees to disk recursively

**Theorem:** Number of cache misses for a search is $O(\log_B N)$.

**Proof.**
cache-oblivious B-tree: (VEB-layout)

• cut tree into subtree at middle level; gives 1 top tree, $\sqrt{N}$ lower trees
• first, write top to disk recursively
• next, write lower trees to disk recursively

Theorem: Number of cache misses for a search is $O(\log_B N)$.

Proof.
Example III: Matrix multiplication
Given $N \times N$ matrices $A$ and $B$, compute $C = A \cdot B$
Given $N \times N$ matrices $A$ and $B$, compute $C = A \cdot B$

IterativeMatrixMult($A, B$)

1. for $i := 1$ to $N$
2. do for $j := 1$ to $N$
3. do for $k := 1$ to $N$
Matrix multiplication: an iterative algorithm

Analysis of number of cache misses: (assume $N > B$)
Analysis of number of cache misses: (assume $N > B$)

IterativeMatrixMult($A, B$)

1. for $i := 1$ to $N$
2. do for $j := 1$ to $N$
3. do for $k := 1$ to $N$
Analysis of number of cache misses: (assume $N > B$)

IterativeMatrixMult($A, B$)
1. for $i := 1$ to $N$
2. do for $j := 1$ to $N$
3. do for $k := 1$ to $N$
Matrix multiplication: an iterative algorithm

Analysis of number of cache misses: (assume $N > B$)

IterativeMatrixMult($A, B$)

1. for $i := 1$ to $N$
2. do for $j := 1$ to $N$
3. do for $k := 1$ to $N$

$O(N/B)$
Matrix multiplication: an iterative algorithm

Analysis of number of cache misses: (assume $N > B$)

IterativeMatrixMult($A, B$)
1. for $i := 1$ to $N$
2. do for $j := 1$ to $N$
3. do for $k := 1$ to $N$
4. do $C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]$ \hspace{1em} $O(N/B)$

When $A$ is stored in row-major order and $B$ is stored in column-major order, then IterativeMatrixMult has $O(N^3/B)$ cache misses.
Matrix multiplication: a recursive algorithm

\[
\begin{array}{cc}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array}
\begin{array}{cc}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{array}
= 
\begin{array}{cc}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{array}
\]

\[A_{11} \cdot B_{11} + A_{12} \cdot B_{21}\]
Matrix multiplication: a recursive algorithm

RecursiveMM\((A, B)\)
1. if \(N = 1\)
2. then return \(A[1, 1] \cdot B[1, 1]\)
3. else \(C_{11} = \text{RecursiveMM}(A_{11}, B_{11}) + \text{RecursiveMM}(A_{12}, B_{21})\)
4. \(C_{12} = \text{RecursiveMM}(A_{11}, B_{12}) + \text{RecursiveMM}(A_{12}, B_{22})\)
5. \(C_{21} = \text{RecursiveMM}(A_{21}, B_{11}) + \text{RecursiveMM}(A_{22}, B_{21})\)
6. \(C_{22} = \text{RecursiveMM}(A_{21}, B_{12}) + \text{RecursiveMM}(A_{22}, B_{22})\)
Analysis of number of cache misses:
Matrix multiplication: a recursive algorithm

Analysis of number of cache misses:

RecursiveMM(A, B)
1. if \( N = 1 \)
2. then return \( A[1, 1] \cdot B[1, 1] \)
3. else \( C_{11} = \text{RecursiveMM}(A_{11}, B_{11}) + \text{RecursiveMM}(A_{12}, B_{21}) \)
4. \( C_{12} = \text{RecursiveMM}(A_{11}, B_{12}) + \text{RecursiveMM}(A_{12}, B_{22}) \)
5. \( C_{21} = \text{RecursiveMM}(A_{21}, B_{11}) + \text{RecursiveMM}(A_{22}, B_{21}) \)
6. \( C_{22} = \text{RecursiveMM}(A_{21}, B_{12}) + \text{RecursiveMM}(A_{22}, B_{22}) \)
Matrix multiplication: a recursive algorithm

Analysis of number of cache misses:

\[ \text{RecursiveMM}(A, B) \]

1. \textbf{if } \( N = 1 \)
2. \textbf{then return } \( A[1, 1] \cdot B[1, 1] \)
3. \textbf{else } \( C_{11} = \text{RecursiveMM}(A_{11}, B_{11}) + \text{RecursiveMM}(A_{12}, B_{21}) \)
4. \( C_{12} = \text{RecursiveMM}(A_{11}, B_{12}) + \text{RecursiveMM}(A_{12}, B_{22}) \)
5. \( C_{21} = \text{RecursiveMM}(A_{21}, B_{11}) + \text{RecursiveMM}(A_{22}, B_{21}) \)
6. \( C_{22} = \text{RecursiveMM}(A_{21}, B_{12}) + \text{RecursiveMM}(A_{22}, B_{22}) \)

\[ T(N) = 8T(N/2) + O(N^2/B) \]
\[ T(\sqrt{M}) = O(M/B) \]
Matrix multiplication: a recursive algorithm

Analysis of number of cache misses:

RecursiveMM(A, B)
1. if $N = 1$
2. then return $A[1, 1] \cdot B[1, 1]$
3. else $C_{11} = \text{RecursiveMM}(A_{11}, B_{11}) + \text{RecursiveMM}(A_{12}, B_{21})$
4. $C_{12} = \text{RecursiveMM}(A_{11}, B_{12}) + \text{RecursiveMM}(A_{12}, B_{22})$
5. $C_{21} = \text{RecursiveMM}(A_{21}, B_{11}) + \text{RecursiveMM}(A_{22}, B_{21})$
6. $C_{22} = \text{RecursiveMM}(A_{21}, B_{12}) + \text{RecursiveMM}(A_{22}, B_{22})$

$T(N) = 8T(N/2) + O(N^2/B)$
$T(\sqrt{M}) = O(M/B)$

If both $A$ and $B$ are stored row-major, then RecursiveMM has $O(N^3/(B\sqrt{M}))$ cache misses.
• fast Fourier transform

• sorting: $O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ cache misses

• priority queues: $O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ cache misses (amortized)

• more (geometric) data structures

• …
bounding-volume hierarchy

data structure for storing objects in $\mathbb{R}^d$
such that objects inside query region can be found quickly
bounding-volume hierarchy

data structure for storing objects in $\mathbb{R}^d$
such that objects inside query region can be found quickly

nodes store bounding box of objects in subtree
bounding-volume hierarchy

data structure for storing objects in $\mathbb{R}^d$
such that objects inside query region can be found quickly

nodes store bounding box of objects in subtree
bounding-volume hierarchy

data structure for storing objects in $\mathbb{R}^d$
such that objects inside query region can be found quickly

nodes store bounding box of objects in subtree
Answering queries
find object intersecting rectangle
Answering queries
find object intersecting rectangle
Answering queries
find object intersecting rectangle
Answering queries
find object intersecting rectangle
Answering queries
find object intersecting rectangle
Answering queries
find object intersecting rectangle
Answering queries
find object intersecting rectangle
R-tree: bounding-volume hierarchy where underlying tree is B-tree

degree = $B$
R-tree: bounding-volume hierarchy where underlying tree is B-tree

There is an R-tree such that any rectangle query can be answered in $O(\sqrt{N/B} + K/B)$ disk accesses, and this is optimal in the worst case.

...and there is a cache-oblivious version [Arge, dB, Haverkort, Yi]
Conclusions

- I/O- and caching behavior crucial for massive data sets
- Algorithms community is now addressing these issues
- I/O-efficient algorithms
  - Many theoretical results, but still a lot of open problems (e.g., graph traversal)
  - Have proven their value for some practical problems
  - Need tuning for hardware, do not optimize caching behavior
- Cache-oblivious algorithms:
  - Ideal in theory: no tuning, good on all cache-levels
  - Some theoretical results, much still open
  - Practical relevance needs further investigation