Accelerating nearest neighbor search on manycore systems

Lawrence Cayton
Max Planck Institute for Intelligent Systems
Why study it?

- NN search is a core subroutine in machine learning (and DB, CG, IR, theory ..)

- But it’s expensive, especially at test time.
Why study it now?
Why study it now?

Research@Intel: The cloud's future is many-core and GPU accelerated

Published about 19 hours ago - by Jon Stokes | Posted in: Uptime

FEATURE STORY

And at the annual Research@Intel day, the chipmaker talked up its plans for the future of the datacenter, and Ars was there to find out what Intel was cooking up in its labs.

READ MORE (2 PAGES)  COMMENT (37)

* from Ars Technica, 15 June 2011.
Why study it now?

• GPUs/multicore CPUs = tremendous power for data analysis

• But unleashing this power requires a fundamental rethinking of algorithms and data structures.

* from Ars Technica, 15 June 2011.
Motivation

NN problem
Motivation

NN problem

(single core) brute force
Motivation

NN problem

(single core) brute force \[\rightarrow\] speed-up \[\rightarrow\] metric data struct

Algorithmic

- Sublinear dependence on \(n\)
- "constant" dependent on intrinsic dimensionality, not extrinsic dimensionality
Aside: extrinsic/intrinsic

Data often only *appears* high-d, but is actually **intrinsically** low-d.

Want algs that scale with the intrinsic dim
Motivation

- (single core) brute force \rightarrow \text{metric data struct}

Algorithmic

- Sublinear dependence on $n$
- “constant” dependent on intrinsic dimensionality, not extrinsic dimensionality
Motivation

NN problem

(single core) brute force
Motivation

NN problem

(single core) brute force

speed-up

(many core) brute force

Structural: BF is trivial to parallelize
Motivation

NN problem

(single core) brute force
Motivation

NN problem

(single core) brute force  \rightarrow  metric data struct

(speed-up)  \rightarrow  speed-up

(many core) brute force  \rightarrow  this work (RBC)

(speedup + speedup)

want structural + algorithmic benefits.
What works on many core?

Matrix multiplication: it’s the operation that gets closest to using all of a processor.

- Many independent operations
- No conditionals
- High memory re-use + regular memory access
NN data structures

Hierarchically decompose space; hopefully will only have to look at a small part

Organize cells into a tree:

Explore using branch-and-bound
On many core?

- Conditional exploration
- Irregular memory accesses / little mem re-use
- Ouch.
Problem setting

Database \( X = \{x_1, x_2, \ldots, x_n\} \)

Query \( q \) (or many queries \( Q \))

Metric \( \rho(\cdot, \cdot) \)

**Goal:** return \( x_i \) minimizing \( \rho(q, x_i) \)

\[ (\forall q \in Q) \]
Brute force search

For each query $q \in Q$, perform a linear scan of $X$;
return the nearest.

Call this procedure $\text{BF}(Q, X)$.

If $I \subset \{1, \ldots, n\}$, $\text{BF}(Q, X[I])$ only considers indices $I$. 
Parallelization of $\mathbf{BF}(Q, X)$

Parallelization of both is incredibly well-studied
Parallelization of $\text{BF}(Q, X)$

1. Compute distances via block decomposition

$$Q = \begin{bmatrix} q_1 \\ \vdots \\ q_m \end{bmatrix}$$

$$X = \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix}$$
Parallelization of $\mathbf{BF}(Q, X)$

1. Compute distances via block decomposition

\[ Q = \begin{bmatrix} q_1 & \vdots & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ q_m & \vdots & \cdots & \vdots \end{bmatrix} \quad X = \begin{bmatrix} x_1 & \cdots & \cdots & x_n \end{bmatrix} \]

2. For each query, do a parallel-reduce on the distances
But..

Work is $O(n)$ per query.
But..

Work is $O(n)$ per query.

This project:

- Reduce the work to roughly $O(\sqrt{n})$ per query
- Maintain the computational structure of $\text{BF}(Q, X)$
Random ball cover - data structure

$r$ random representatives

Ball around representatives containing $s$ points
Random ball cover - data structure

Notation: $L_r$ - indices of points owned by rep $r$. 
One-shot search algorithm

1. compute nearest representative
One-shot search algorithm cont.

2. find nearest point within set covered by nearest representative
One-shot algorithm: restatement

Call $\text{BF}(q, R)$; get rep $r$ back.

Call $\text{BF}(q, X[L_r])$.

\textit{i.e. two brute force searches}

(later, we’ll see that each is roughly $O(\sqrt{n})$)
Exact search algorithm

1. compute nearest representative

(same as before)
Exact search algorithm

2. prune out as many balls as possible
Exact search algorithm

3. Search the rest and return the nearest.
Exact search restatement

Call $\textbf{BF}(q, R)$; get rep $r$ back.

Compute lists $L_1, \ldots, L_t$ that can not be pruned.

Call $\textbf{BF}(q, X[L_1 \cup \cdots \cup L_t])$. 
Theory

Both algs have

- $O(\sqrt{n})$ dependence on the data
- some dependence on the growth rate $c,$

where $c \approx 2^{\text{intrinsic dim}}.$
Exact search alg

Guaranteed to find the exact NN; but how long does it take?

Data structure details:

- Each rep $r$ chosen independently w.p. $p$.
- Each $x \in X$ assigned to nearest $r$.

( think of $p \approx \frac{c}{\sqrt{n}}$ )
First step has expected complexity $1/p$.

Third step: want to bound $|L_1 \cup \cdots \cup L_t|$.
Exact search alg

Let $\gamma = \rho(q, r_q)$ (dist to $q$’s NN among $R$).

How many points are in $B(q, \gamma)$?

In expectation, about $1/p$. 
Recall that all relevant reps $r$ sat $\rho(q, r) \leq 3\gamma$.

Can show that the NN of $q$ must lie in $B(q, 7\gamma)$. 
Exact search alg

Setting \( p = O(c^{3/2}/\sqrt{n}) \),

and applying the growth rate condition,

get bound on \( |B(q, 7\gamma)| \)

the expected run time is \( O(c^{3/2}\sqrt{n}) \).
One shot alg

Recall:

1. Call $\textbf{BF}(q, R)$; get $r_q$.

2. Call $\textbf{BF}(q, X[L])$. 
One shot alg

Recall:

1. Call $\text{BF}(q, R)$; get $r_q$.
2. Call $\text{BF}(q, X[L])$.

Set $n_r = s = c\sqrt{n} \cdot \sqrt{\ln \frac{1}{\delta}}$.

Then the one-shot alg is correct w.p. $\geq 1 - \delta$. 

Experiments on 48 cores

Experiments show two things:

1. The RBC search alg reduces the work for NN (supports the theory)

2. It parallelizes effectively (supports the design choices)
## Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Num pts</th>
<th>Dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bio</td>
<td>200k</td>
<td>74</td>
</tr>
<tr>
<td>Covertype</td>
<td>500k</td>
<td>54</td>
</tr>
<tr>
<td>Physics</td>
<td>100k</td>
<td>78</td>
</tr>
<tr>
<td>Robot</td>
<td>2M</td>
<td>21</td>
</tr>
<tr>
<td>TinyIm</td>
<td>10M</td>
<td>4-32</td>
</tr>
</tbody>
</table>
Exact search results

- bio
- cov
- phy
- robot
- tiny4
- tiny8
- tiny16
- tiny32
## Actual times for 10k queries

<table>
<thead>
<tr>
<th>Data</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bio</td>
<td>.4s</td>
</tr>
<tr>
<td>Cov</td>
<td>.4s</td>
</tr>
<tr>
<td>Phy</td>
<td>.3s</td>
</tr>
<tr>
<td>Robot</td>
<td>1.2s</td>
</tr>
<tr>
<td>Tiny4</td>
<td>.7s</td>
</tr>
<tr>
<td>Tiny8</td>
<td>.8s</td>
</tr>
<tr>
<td>Tiny16</td>
<td>3.0s</td>
</tr>
<tr>
<td>Tiny32</td>
<td>7.5s</td>
</tr>
</tbody>
</table>
One-shot search

The parameter allows you to trade-off between speed and quality.

Error measure: rank of returned point. 
* e.g. rank-0 is exact NN, rank-1 is 2nd NN, ..
One-shot search results

bio

cov

phy

robot

tiny4 and tiny8

tiny16 and tiny32
## GPU results

<table>
<thead>
<tr>
<th>Data</th>
<th>Speedup (GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bio</td>
<td>38.1</td>
</tr>
<tr>
<td>Covertype</td>
<td>94.6</td>
</tr>
<tr>
<td>Physics</td>
<td>19.0</td>
</tr>
<tr>
<td>Robot</td>
<td>53.2</td>
</tr>
<tr>
<td>TinyIm4</td>
<td>188.4</td>
</tr>
</tbody>
</table>
# Cover tree comparison

<table>
<thead>
<tr>
<th>Data</th>
<th>Cover Tree</th>
<th>RBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bio</td>
<td>18.9</td>
<td>6.4</td>
</tr>
<tr>
<td>Covertype</td>
<td>0.4</td>
<td>1.1</td>
</tr>
<tr>
<td>Physics</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>Robot</td>
<td>4.6</td>
<td>5.1</td>
</tr>
<tr>
<td>Tiny4</td>
<td>0.5</td>
<td>1.2</td>
</tr>
<tr>
<td>Tiny8</td>
<td>14.6</td>
<td>3.3</td>
</tr>
<tr>
<td>Tiny16</td>
<td>178.9</td>
<td>25.1</td>
</tr>
<tr>
<td>Tiny32</td>
<td>387.0</td>
<td>67.9</td>
</tr>
</tbody>
</table>
Conclusion

• Simple, high performance method
• Broadly applicable
• Theoretically sound
• Good implementations available