Multicore Triangle Computations Without Tuning

Julian Shun and Kanat Tangwongsan

Presentation is based on paper published in International Conference on Data Engineering (ICDE), 2015
Triangle Computations

- Triangle Counting
  Count = 3

- Other variants:
  - Triangle listing
  - Local triangle counting/clustering coefficients
  - Triangle enumeration
  - Approximate counting
  - Analogs on directed graphs

- Numerous applications…
  - Social network analysis, Web structure, spam detection, outlier detection, dense subgraph mining, 3-way database joins, etc.

Need fast triangle computation algorithms!
Sequential Triangle Computation Algorithms

V = # vertices  E = # edges

• Sequential algorithms for exact counting/listing
  • Naïve algorithm of trying all triplets
    \(O(V^3)\) work
  • Node-iterator algorithm [Schank]
    \(O(VE)\) work
  • Edge-iterator algorithm [Itai-Rodeh]
    \(O(VE)\) work
  • Tree-lister [Itai-Rodeh], forward/compact-forward [Schank-Wagner, Lapaty]
    \(O(E^{1.5})\) work

• Sequential algorithms via matrix multiplication
  • \(O(V^{2.37})\) work compute \(A^3\), where \(A\) is the adjacency matrix
  • \(O(E^{1.41})\) work [Alon-Yuster-Zwick]
  • These require superlinear space
# Sequential Triangle Computation Algorithms

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Description</th>
<th>Source: “Algorithmic Aspects of Triangle-Based Network Analysis”, Dissertation by Thomas Schank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(m^{2\gamma+1})$</td>
<td>ayz using fast matrix-multiplication</td>
<td></td>
</tr>
<tr>
<td>$O(n^\gamma)$</td>
<td>fast matrix-multiplication</td>
<td></td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>matrix-multiplication</td>
<td></td>
</tr>
<tr>
<td>$O(m^{3/2})$</td>
<td>core</td>
<td></td>
</tr>
<tr>
<td>$O(d_{max} \cdot m)$</td>
<td>forward-hashed (compact) forward hashed</td>
<td></td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>try-all</td>
<td></td>
</tr>
</tbody>
</table>

**What about parallel algorithms?**
Parallel Triangle Computation Algorithms

- Most designed for distributed memory
  - MapReduce algorithms [Cohen ’09, Suri-Vassilvitskii ‘11, Park-Chung ‘13, Park et al. ‘14]
  - MPI algorithms [Arifuzzaman et al. ‘13, Graphlab]

- What about shared-memory multicore?
  - Multicores are everywhere!
  - Node-iterator algorithm [Green et al. ‘14]
    - $O(VE)$ work in worst case

- Can we obtain an $O(E^{1.5})$ work shared-memory multicore algorithm?
Triangle Computation: Challenges for Shared Memory Machines

1. Irregular computation

2. Deep memory hierarchy
External-Memory and Cache-Oblivious Triangle Computation

- All previous algorithms are sequential
- External-memory (cache-aware) algorithms
  - Natural-join
    \[ O\left(\frac{E^3}{M^2 B}\right) \] I/O’s
  - Node-iterator [Dementiev ’06]
    \[ O\left(\frac{E^{1.5}}{B} \log_{M/B}(E/B)\right) \] I/O’s
  - Compact-forward [Menegola ‘10]
    \[ O(E + \frac{E^{1.5}}{B}) \] I/O’s
  - [Chu-Cheng ’11, Hu et al. ‘13]
    \[ O\left(\frac{E^2}{MB} + \#\text{triangles}/B\right) \] I/O’s
- External-memory and cache-oblivious
  - [Pagh-Silvestri ‘14]
    \[ O\left(\frac{E^{1.5}}{(M^{0.5}B)}\right) \] I/O’s or cache misses

- Parallel cache-oblivious algorithms?
Our Contributions

1. **Parallel Cache-Oblivious Triangle Counting Algs**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth</th>
<th>Cache Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC-Merge</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^2 E)$</td>
<td>$O(E + E^{1.5}/B)$</td>
</tr>
<tr>
<td>TC-Hash</td>
<td>$O(V \log V + \alpha E)$</td>
<td>$O(\log^2 E)$</td>
<td>$O(\text{sort}(V) + \alpha E)$</td>
</tr>
<tr>
<td>Par. Pagh-Silvestri</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^3 E)$</td>
<td>$O(E^{1.5}/(M^{0.5} B))$</td>
</tr>
</tbody>
</table>

$V = \#\ \text{vertices} \quad E = \#\ \text{edges} \quad \alpha = \text{arboricity (at most } E^{0.5})$

$M = \text{cache size} \quad B = \text{line size} \quad \text{sort}(n) = (n/B) \log_{M/B}(n/B)$

2. **Extensions to Other Triangle Computations:**
   - Enumeration, Listing, Local Counting/Clustering Coefficients,
   - Approx. Counting, Variants on Directed Graphs

3. **Extensive Experimental Study**
Sequential Triangle Counting (Exact)

*(Forward/compact-forward algorithm)*

1. Rank vertices by degree (sorting)
   Return $A[v]$ for all $v$ storing higher ranked neighbors

2. For each vertex $v$:
   - For each $w$ in $A[v]$:
     - $\text{count} += \text{intersect}(A[v], A[w])$

Gives all triangles $(v, w, x)$ where $\text{rank}(v) < \text{rank}(w) < \text{rank}(x)$

Work = $O(E^{1.5})$  
[Schank-Wagner ‘05, Latapy ‘08]
Proof of $O(E^{1.5})$ work bound when intersect uses merging

1. Rank vertices by degree (sorting)
   Return $A[v]$ for all $v$ storing higher ranked neighbors

2. for each vertex $v$:
   for each $w$ in $A[v]$:
   count += intersect($A[v], A[w]$)

- Step 1: $O(E + V \log V)$ work
- Step 2:
  - For each edge $(v,w)$, intersect does $O(d^+(v) + d^+(w))$ work
  - For all $v$, $d^+(v) \leq E^{0.5}$
    - If $d^+(v) > E^{0.5}$, each of its higher ranked neighbors also have degree $> E^{0.5}$ and total number of directed edges $> E$, a contradiction
  - Total work $= E \times O(E^{0.5}) = O(E^{1.5})$
Parallel Triangle Counting (Exact)

Step 1
Work = O(E+V log V)
Depth = O(log^2 V)
Cache = O(E+sort(V))

Parallel sort and filter

Rank vertices by degree (sorting)
Return $A[v]$ for all $v$ storing higher ranked neighbors

Parallel reduction

Parallel merge (TC-Merge) or Parallel hash table (TC-Hash)

Safe to run all in parallel
TC-Merge and TC-Hash Details

Parallel reduction

\begin{align*}
\text{parallel}_\text{for each vertex } v: \\
\text{parallel}_\text{for each } w \text{ in } A[v]: \\
\quad \text{count }+&= \text{intersect}(A[v], A[w])
\end{align*}

\begin{itemize}
  \item **TC-Merge**
    \begin{itemize}
      \item Preprocessing: sort adjacency lists
      \item Intersect: use a parallel and cache-obliviuous merge based on divide-and-conquer [Blelloch et al. ‘10, Blelloch et al. ‘11]
    \end{itemize}
  \item **TC-Hash**
    \begin{itemize}
      \item Preprocessing: for each vertex, create parallel hash table storing edges [Shun-Blelloch ‘14]
      \item Intersect: scan smaller list, querying hash table of larger list in parallel
    \end{itemize}
\end{itemize}

Step 2: TC-Merge
Work = \(O(E^{1.5})\)
Depth = \(O(\log^2 E)\)
Cache = \(O(E+E^{1.5}/B)\)

Step 2: TC-Hash
Work = \(O(\alpha E)\)
Depth = \(O(\log E)\)
Cache = \(O(\alpha E)\)

(\(\alpha = \text{arboricity (at most } E^{0.5})\))

Parallel merge (TC-Merge) or Parallel hash table (TC-Hash)
## Comparison of Complexity Bounds

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth</th>
<th>Cache Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC-Merge</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^2 E)$</td>
<td>$O(E + E^{1.5}/B)$ (<em>oblivious</em>)</td>
</tr>
<tr>
<td>TC-Hash</td>
<td>$O(V \log V + \alpha E)$</td>
<td>$O(\log^2 E)$</td>
<td>$O(\text{sort}(V) + \alpha E)$ (<em>oblivious</em>)</td>
</tr>
<tr>
<td>Par. Pagh-Silvestri</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^3 E)$</td>
<td>$O(E^{1.5}/(M^{0.5} B))$ (<em>oblivious</em>)</td>
</tr>
<tr>
<td>Chu-Cheng ‘11, Hu et al. ‘13</td>
<td>$O(E \log E + E^2/M + \alpha E)$</td>
<td></td>
<td>$O(E^2/(MB) + \text{#triangles}/B)$ (<em>aware</em>)</td>
</tr>
<tr>
<td>Pagh-Silvestri ‘14</td>
<td>$O(E^{1.5})$</td>
<td></td>
<td>$O(E^{1.5}/(M^{0.5} B))$ (<em>oblivious</em>)</td>
</tr>
<tr>
<td>Green et al. ‘14</td>
<td>$O(VE)$</td>
<td>$O(\log E)$</td>
<td></td>
</tr>
</tbody>
</table>

$V = \#$ vertices  \hspace{1cm} $E = \#$ edges  \hspace{1cm} $\alpha = \text{arboricity (at most } E^{0.5})$

$M = \text{cache size}  \hspace{1cm} B = \text{line size}  \hspace{1cm} \text{sort}(n) = (n/B) \log_{M/B}(n/B)$
Our Contributions

1. Parallel Cache-Oblivious Triangle Counting Algs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth</th>
<th>Cache Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC-Merge</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^2 E)$</td>
<td>$O(E + E^{1.5}/B)$</td>
</tr>
<tr>
<td>TC-Hash</td>
<td>$O(V \log V + \alpha E)$</td>
<td>$O(\log^2 E)$</td>
<td>$O(\text{sort}(V) + \alpha E)$</td>
</tr>
<tr>
<td>Par. Pagh-Silvestri</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^3 E)$</td>
<td>$O(E^{1.5}/(M^{0.5} B))$</td>
</tr>
</tbody>
</table>

$V = \#$ vertices  $E = \#$ edges  $\alpha =$ arboricity (at most $E^{0.5}$)
$M = \text{cache size}$  $B = \text{line size}$  $\text{sort}(n) = (n/B) \log_{M/B}(n/B)$

2. Extensions to Other Triangle Computations:
   - Enumeration, Listing, Local Counting/Clustering Coefficients,
   - Approx. Counting, Variants on Directed Graphs

3. Extensive Experimental Study
Extensions of Exact Counting Algorithms

• Triangle enumeration
  • Call \textbf{emit} function whenever triangle is found
  • \textbf{Listing}: add to hash table to list; return contents at the end
  • \textbf{Local counting/clustering coefficients}: atomically increment count of three triangle endpoints

• Directed triangle counting/enumeration
  • Keep separate counts for different types of triangles

• Approximate counting
  • Use colorful triangle sampling scheme to create smaller sub-graph \cite{Pagh-Tsourakakis12}
  • Run TC-Merge or TC-Hash on sub-graph with pE edges ($0 < p < 1$) and return $\#\text{triangles}/p^2$ as estimate
## Approximate Counting

- **Colorful triangle counting** [Pagh-Tsourakakis ’12]
  
  - **Sampling rate**: $0 < p < 1$
  
  - Assign random color in $\{1, \ldots, 1/p\}$ to each vertex
  
  - Sampling: Keep edges whose endpoints have the same color
  
  - Run exact triangle counting on sampled graph, return $\Delta_{\text{sampled}}/p^2$

### Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parallel scan</td>
<td>$O(E)$, $O(\log E)$, $O(E/B)$</td>
</tr>
<tr>
<td>2</td>
<td>Parallel filter</td>
<td>$O((pE)^{1.5})$, $O(\log^2 E)$, $O(pE+(pE)^{1.5}/B)$</td>
</tr>
<tr>
<td>3</td>
<td>Use TC-Merge or TC-Hash</td>
<td>$O(V \log V + \alpha pE)$, $O(\log E)$, $O(\text{sort}(V)+p\alpha E)$</td>
</tr>
</tbody>
</table>

Expected # edges = $pE$
Our Contributions

1. **Parallel Cache-Oblivious Triangle Counting Algs**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth</th>
<th>Cache Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC-Merge</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^2 E)$</td>
<td>$O(E + E^{1.5}/B)$</td>
</tr>
<tr>
<td>TC-Hash</td>
<td>$O(V \log V + \alpha E)$</td>
<td>$O(\log^2 E)$</td>
<td>$O(\text{sort}(V) + \alpha E)$</td>
</tr>
<tr>
<td>Par. Pagh-Silvestri</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^3 E)$</td>
<td>$O(E^{1.5}/(M^{0.5} B))$</td>
</tr>
</tbody>
</table>

$V = \# \text{ vertices}$

$E = \# \text{ edges}$

$\alpha = \text{arboricity (at most } E^{0.5})$

$M = \text{cache size}$

$B = \text{line size}$

$\text{sort}(n) = (n/B) \log_{M/B}(n/B)$

2. **Extensions to Other Triangle Computations:**
   - Enumeration, Listing, Local Counting/Clustering Coefficients,
   - Approx. Counting, Variants on Directed Graphs

3. **Extensive Experimental Study**
Experimental Setup

- Implementations using Intel Cilk Plus
- 40-core Intel Nehalem machine (with 2-way hyper-threading)
  - 4 sockets, each with 30MB shared L3 cache, 256KB private L2 caches
- Sequential TC-Merge as baseline (faster than existing sequential implementations)
- Other multicore implementations: Green et al. and GraphLab
- Our parallel Pagh-Silvestri algorithm was not competitive
- Variety of real-world and artificial graphs
Both TC-Merge and TC-Hash scale well with # of cores:

- **LiveJournal**
  - 4M vtxes, 34.6M edges
  - ~ 27x

- **Orkut**
  - 3M vtxes, 117M edges
  - ~ 48x
40-core (with hyper-threading) Performance

- TC-Merge always faster than TC-Hash (by 1.3—2.5x)
- TC-Merge always faster than Green et al. or GraphLab (by 2.1—5.2x)
Why is TC-Merge faster than TC-Hash?

- TC-Hash less cache-efficient than TC-Merge
- Running time more correlated with cache misses than work
Comparison to existing counting algs.

Twitter graph (41M vertices, 1.2B undirected edges, 34.8B triangles)

• **Yahoo graph** (1.4B vertices, 6.4B edges, 85.8B triangles) on 40 cores: **TC-Merge takes 78 seconds**
  – Approximate counting algorithm achieves **99.6% accuracy in 9.1 seconds**
## Shared vs. distributed memory costs

- Amazon EC2 pricing
  - Captures purchasing costs, maintenance/operating costs, energy costs

<table>
<thead>
<tr>
<th>Triangle Counting (Twitter)</th>
<th>Our algorithm</th>
<th>GraphLab</th>
<th>GraphLab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time</td>
<td>0.932 min</td>
<td>3 min</td>
<td>1.5 min</td>
</tr>
<tr>
<td>Machine</td>
<td>40-core (256 GB memory)</td>
<td>40-core (256 GB memory)</td>
<td>64 x 16-core</td>
</tr>
<tr>
<td>Approx. EC2 pricing</td>
<td>&lt; $4/hour</td>
<td>&lt; $4/hour</td>
<td>64 x $0.928/hour</td>
</tr>
<tr>
<td>Overall cost</td>
<td>&lt; $0.062</td>
<td>&lt; $0.2</td>
<td>$1.49</td>
</tr>
</tbody>
</table>
Approximate counting

\[ \frac{T_{\text{approx}}}{T_{\text{exact}}} \]

<table>
<thead>
<tr>
<th>p=1/25</th>
<th>Accuracy</th>
<th>(T_{\text{approx}})</th>
<th>(T_{\text{approx}}/T_{\text{exact}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orkut (V=3M, E=117M)</td>
<td>99.8%</td>
<td>0.067sec</td>
<td>0.035</td>
</tr>
<tr>
<td>Twitter (V=41M, E=1.2B)</td>
<td>99.9%</td>
<td>2.4sec</td>
<td>0.043</td>
</tr>
<tr>
<td>Yahoo (V=1.4B, E=6.4B)</td>
<td>99.6%</td>
<td>9.1sec</td>
<td>0.117</td>
</tr>
</tbody>
</table>
Conclusion

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth</th>
<th>Cache Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC-Merge</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^2 E)$</td>
<td>$O(E + E^{1.5}/B)$</td>
</tr>
<tr>
<td>TC-Hash</td>
<td>$O(V \log V + \alpha E)$</td>
<td>$O(\log^2 E)$</td>
<td>$O(\text{sort}(V) + \alpha E)$</td>
</tr>
<tr>
<td>Par. Pagh-Silvestri</td>
<td>$O(E^{1.5})$</td>
<td>$O(\log^3 E)$</td>
<td>$O(E^{1.5}/(M^{0.5} B))$</td>
</tr>
</tbody>
</table>

- Simple multicore algorithms for triangle computations are provably work-efficient, low-depth, and cache-efficient.
- Implementations require no load-balancing or tuning for cache.
- Experimentally outperforms existing multicore and distributed algorithms.
- Future work: Design a practical parallel algorithm achieving $O(E^{1.5}/(M^{0.5} B))$ cache complexity.