Parallel Peeling Algorithms

Justin Thaler, Yahoo Labs
Joint Work with:
Michael Mitzenmacher, Harvard University
Jiayang Jiang
The Peeling Paradigm

- Many important algorithms for a wide variety of problems can be modeled in the same way.

- Start with a (random) hypergraph $G$.
  - While there exists a node $v$ of degree less than $k$:
    - Remove $v$ and all incident edges.

- The remaining graph is called the **k-core** of $G$.
  - $k=2$ in most applications.

- Typically, the algorithm “succeeds” if the k-core is empty.
  - To ensure “success”, data structure should be designed large enough so that the k-core of $G$ is empty w.h.p.

- Typically yields simple, greedy algorithms running in linear time.
The peeling process when $k=2$
The peeling process when $k=2$
The peeling process when $k=2$
The peeling process when $k=2$
The peeling process when $k=2$
The peeling process when $k=2$
The peeling process when $k=2$
The peeling process when $k=2$
The peeling process when $k=2$
Example Algorithms
Example 1: Sparse Recovery Algorithms

- Consider data streams that insert and delete a lot of items.
  - Flows through a router, people entering/leaving a building.
- Sparse Recovery problem: list all items with non-zero frequency.
- Want listing not at all times, but at “reasonable” or “off-peak” times, when working set size is bounded.
  - If we do $N$ insertions, then $N-M$ deletions, and want a list at the end, we need to list $M$ items.
- Data structure size should be proportional to $M$, not to $N$!
  - Proportional to size you want to be able to list, not number of items your system has to handle.
- Central primitive used in more complicated streaming algorithms.
  - E.g. $L_0$ sampling, which is in turn used to solve problems on dynamic graph streams (see previous talk).
Example 1: Sparse Recovery Algorithms

- For simplicity, assume that when listing occurs, no item has frequency more than 1.
Example 1: Sparse Recovery Algorithms

- **Sparse Recovery Algorithm**: Invertible Bloom Lookup Tables (IBLTs) [Goodrich, Mitzenmacher]

Each stream item hashed to r cells (using r different hash functions)

Insert(x): For each of the j cells that x is hashed to:
- Add key to KeySum
- Increment Count

Delete(x): For each of the j cells x is hashed to:
- Subtract key from keysum
- Decrement Count
Listing Algorithm: Peeling

- Call a cell “pure” if its count equals 1.
- While there exists a pure cell:
  - Output $x=\text{keySum}$ of the cell.
  - Call Delete($x$) on the IBLT.
Listing Algorithm: Peeling

- Call a cell “pure” if its count equals 1.
- While there exists a pure cell:
  - Output $x = \text{keySum}$ of the cell.
  - Call Delete($x$) on the IBLT.
- To handle frequencies that are larger than 1, add a checksum field to each cell (details omitted).
Listing Algorithm: Peeling

- Call a cell “pure” if its count equals 1.
- While there exists a pure cell:
  - Output \( x = \text{keySum} \) of the cell.
  - Call \( \text{Delete}(x) \) on the IBLT.
- To handle frequencies that are larger than 1, add a checksum field to each cell (details omitted).
- Listing peeling to 2-core on the hypergraph \( G \) where:
  - Cells \( \leftrightarrow \) vertices of \( G \).
  - Items in IBLT \( \leftrightarrow \) hyperedges of \( G \).
  - \( G \) is \( r \)-uniform (each edge has \( r \) vertices, one for each cell the item is hashed to).
How Many Cells Does an IBLT Need to Guarantee Successful Listing?

- Consider a random $r$-uniform hypergraph $G$ with $n$ nodes and $m = c*n$ edges.
  - i.e., each edge has $r$ vertices, chosen uniformly at random from $[n]$ without repetition.

- Known fact: Appearance of a non-empty $k$-core obeys a sharp threshold.
  - For some constant $c_{k,r}$, when $m < c_{k,r} n$, the $k$-core is empty with probability $1-o(1)$.
  - When $m > c_{k,r} n$, the $k$-core of $G$ is non-empty with probability $1-o(1)$.
  - Implication: to successfully list a set of size $M$ with probability $1-o(1)$, the IBLT needs roughly $M/c_{k,r}$ cells.

- E.g. $c_{2,3} \approx 0.818$, $c_{2,4} \approx 0.772$, $c_{3,3} \approx 1.553$. 
How Many Cells Does an IBLT Need to Guarantee Successful Listing?

- Consider a random $r$-uniform hypergraph $G$ with $n$ nodes and $m = cn$ edges. i.e., each edge has $r$ vertices, chosen uniformly at random from $[n]$ without repetition.
- Known fact: Appearance of a non-empty $k$-core obeys a sharp threshold.
  - For some constant $c_{k,r}$, when $m < c_{k,r}n$, the $k$-core is empty with probability $1 - o(1)$.
  - When $m > c_{k,r}n$, the $k$-core of $G$ is non-empty with probability $1 - o(1)$.
  - Implication: to successfully list a set of size $M$ with probability $1 - o(1)$, the IBLT needs roughly $M/c_{k,r}$ cells.
- E.g. $c_{2,3} \approx 0.818$, $c_{2,4} \approx 0.772$, $c_{3,3} \approx 1.553$.
- In general:
  $$c_{k,r}^* = \min_{x > 0} \frac{x}{r(1 - e^{-x} \sum_{j=0}^{k-2} \frac{x^j}{j!})^{r-1}}.$$
Other Examples of Peeling Algorithms

- Low-Density Parity Check Codes for Erasure Channel.
  - [Luby, Mitzenmacher, Shokrollah, Spielman]
- Biff codes (directly use IBLTs).
  - [Mitzenmacher and Varghese]
- k-wise independent hash families with $O(1)$ evaluation time.
  - [Siegel]
- Sparse FFT algorithms.
  - [Hassanieh et al.]
- Cuckoo hashing.
  - [Pagh and Rodler]
- Pure literal rule for computing satisfying assignments of random CNFs.
  - [Franco] [Mitzenmacher] [Molloy] [many others].
Parallel Peeling Algorithms
Our Goal: Parallelize These Peeling Algorithms

- Recall: the aforementioned algorithms are equivalent to peeling a random hypergraph $G$ to its $k$-core.
- There is a brain dead way to parallelize the peeling process.
  - For each node $v$ in parallel:
    - Check if $v$ has degree less than $k$.
    - If so, remove $v$ and its incident hyperedges.
- Key question: how many rounds of peeling are required to find the $k$-core?
- Algorithm is simple, analysis is tricky.
Main Result

- Two behaviors:
  - Parallel peeling completes in $O(\log \log n)$ rounds if the edge density $c$ is “below the threshold” $c_{k,r}$.
  - Parallel peeling requires $\Omega(\log n)$ rounds if the edge density $c$ is “above the threshold” $c_{k,r}$.
- This is great!
  - Most peeling uses the goal is to be below the threshold.
  - So “nature” is helping us by making parallelization fast.
  - Implies $\text{poly} (\log \log n)$ time, $O(\text{n \ poly}(\log \log n))$ work, parallel algorithms for listing elements in an IBLT, decoding LDPC codes, etc.
Precise Upper Bound

**Theorem 1.** Let \( k, r \geq 2 \) with \( k + r \geq 5 \), and let \( c \) be a constant. With probability \( 1 - o(1) \), the parallel peeling process for the \( k \)-core in a random hypergraph \( G_{n, cn}^r \) with edge density \( c \) and \( r \)-ary edges terminates after
\[
\frac{1}{\log((k-1)(r-1))} \log \log n + O(1)
\] rounds when \( c < c_k^* \).

**Theorem 2.** Let \( k, r \geq 2 \) with \( k + r \geq 5 \), and let \( c \) be a constant. With probability \( 1 - o(1) \), the parallel peeling process for the \( k \)-core in a random hypergraph \( G_{n, cn}^r \) with edge density \( c \) and \( r \)-ary edges requires
\[
\frac{1}{\log((k-1)(r-1))} \log \log n - O(1)
\] rounds to terminate when \( c < c_k^* \).

**Summary:** The right factor in front of the \( \log \log n \) is \( 1 / (\log(k-1)(r-1)) \)
(tight up to an additive constant).
**Lower Bound**

**Theorem 3.** Let \( r \geq 3 \) and \( k \geq 2 \). With probability \( 1 - o(1) \), the peeling process for the \( k \)-core in \( G_{n, cn}^r \) terminates after \( \Omega(\log n) \) rounds when \( c > c_{k, r}^* \).

Summary: \( \Omega(\log n) \) lower bound matches an earlier \( O(\log n) \) upper bound due to [Achlioptas and Molloy, 2013].
Proof Sketch for Upper Bound

- Let $\lambda_i$ denote the probability a given vertex $v$ survives $i$ rounds of peeling.
- Claim: $\lambda_{i+1} \leq (C\lambda_i)^{(k-1)(r-1)}$ for some constant $C$.
  - Suggests $\lambda_i \ll 1/n$ after about $1/\log((k-1)(r-1))*\log\log n$ rounds.
  - A related argument shows that $\lambda_i \leq 1/(2C)$ after $O(1)$ rounds, and after that point the claim implies that $\lambda_i$ falls doubly-exponentially quickly.
Proof Sketch for Upper Bound

- Let $\lambda_i$ denote the probability a given vertex $v$ survives $i$ rounds of peeling.
- Claim: $\lambda_{i+1} \leq (C\lambda_i)^{(k-1)(r-1)}$ for some constant $C$.
- **Very** crude sketch of the Claim’s plausibility:
  - Node $v$ survives round $i+1$ only if it has (at least) $k$ incident edges $e_1 \ldots e_k$ that survive round $i$.
  - Fix a $k$-tuple of edges $e_1 \ldots e_k$ incident to $v$.
  - Assume no node other than $v$ appears in more than one of these edges.
  - Then there are $k(r-1)$ distinct nodes other than $v$ appearing in these edges.
  - The edges all survive round $i$ only if all $k(r-1)$ of these nodes survive round $i$.
  - Let’s pretend that the survival of these nodes are independent events.
  - Then the probability all nodes survive round $i$ is roughly $\lambda_i^{k(r-1)}$.
  - Finally, union bound over all $k$-tuples of edges incident to $v$. 
Simulation Results

- Results from simulations of parallel peeling process on random 4-uniform hypergraphs with $n$ nodes and $c \times n$ edges using $k = 2$.
- Averaged over 1000 trials.
- Recall that $c_{2,4} \approx 0.772$. 

<table>
<thead>
<tr>
<th>$n$</th>
<th>Failed</th>
<th>Rounds</th>
<th>Failed</th>
<th>Rounds</th>
<th>Failed</th>
<th>Rounds</th>
<th>Failed</th>
<th>Rounds</th>
<th>Failed</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0</td>
<td>12.504</td>
<td>0</td>
<td>23.352</td>
<td>1000</td>
<td>17.037</td>
<td>1000</td>
<td>10.773</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20000</td>
<td>0</td>
<td>12.594</td>
<td>0</td>
<td>23.433</td>
<td>1000</td>
<td>19.028</td>
<td>1000</td>
<td>11.928</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40000</td>
<td>0</td>
<td>12.791</td>
<td>0</td>
<td>23.343</td>
<td>1000</td>
<td>20.961</td>
<td>1000</td>
<td>12.992</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80000</td>
<td>0</td>
<td>12.939</td>
<td>0</td>
<td>23.372</td>
<td>1000</td>
<td>22.959</td>
<td>1000</td>
<td>14.104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160000</td>
<td>0</td>
<td>12.983</td>
<td>0</td>
<td>23.421</td>
<td>1000</td>
<td>25.066</td>
<td>1000</td>
<td>15.005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320000</td>
<td>0</td>
<td>13.000</td>
<td>0</td>
<td>23.491</td>
<td>1000</td>
<td>27.089</td>
<td>1000</td>
<td>16.305</td>
<td></td>
<td></td>
</tr>
<tr>
<td>640000</td>
<td>0</td>
<td>13.000</td>
<td>0</td>
<td>23.564</td>
<td>1000</td>
<td>29.281</td>
<td>1000</td>
<td>17.334</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1280000</td>
<td>0</td>
<td>13.000</td>
<td>0</td>
<td>23.716</td>
<td>1000</td>
<td>31.037</td>
<td>1000</td>
<td>18.499</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2560000</td>
<td>0</td>
<td>13.000</td>
<td>0</td>
<td>23.840</td>
<td>1000</td>
<td>33.172</td>
<td>1000</td>
<td>19.570</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Refined Result: Mind the Gap

**Theorem 7.1.** Let $v = |c^*_{k,r} - c|$ for constant $c$ with $c < c_{k,r}$.
With probability $1 - o(1)$, peeling in $G^r_{n, cn}$ requires $\Theta(1/v) + \frac{1}{\log((k-1)(r-1))} \log \log n$ rounds when $c$ is below the threshold density $c^*_{k,r}$.

Summary: below the threshold, the additive term is $\Theta(1/\sqrt{|\text{gap}|})$. This can be more important than the $\log \log n$ term if the edge density is close to the threshold!
Plots show expected progress of the peeling process as a function of the round $i$, for values of the edge density $c$ approaching the threshold value of $c_{2,4} \approx 0.772$. 
Refined Analysis: Mind the Gap

• Analysis shows that peeling process falls into three “stages”.
  • First stage: the fraction of surviving nodes falls very quickly as a function of the rounds until it gets close to a certain key value $x^*$.
  • Second stage: $\Theta(1/\sqrt{|\text{gap}|})$ rounds are required to go from “close” to $x^*$ to “significantly below” $x^*$.
  • Third stage: the analysis of the basic upper bound kicks in, and the fraction of surviving nodes falls doubly-exponentially quickly.
Implementation Issues
GPU Experimental Results

<table>
<thead>
<tr>
<th>Table Load</th>
<th>No. Table Cells</th>
<th>% Recovered</th>
<th>GPU Recovery Time</th>
<th>Serial Recovery Time</th>
<th>GPU Insert Time</th>
<th>Serial Insert Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.75</td>
<td>16.8 million</td>
<td>100%</td>
<td>0.33 s</td>
<td>6.37 s</td>
<td>0.31 s</td>
<td>3.91 s</td>
</tr>
<tr>
<td>0.83</td>
<td>16.8 million</td>
<td>50.1%</td>
<td>0.42 s</td>
<td>3.64 s</td>
<td>0.35 s</td>
<td>4.34 s</td>
</tr>
</tbody>
</table>

Table 3: Results of our parallel and serial IBLT implementations with \( r = 3 \) hash functions. The table load refers to the ratio of the number of items in the IBLT to the number of cells in the IBLT.
Recall: IBLTs

Each stream item hashed to \( r \) cells (using \( r \) different hash functions)

**Insert\((x)\):** For each of the \( j \) cells that \( x \) is hashed to:
- Add key to KeySum
- Increment Count

**Delete\((x)\):** For each of the \( j \) cells \( x \) is hashed to:
- Subtract key from keysum
- Decrement Count
Recall: IBLT Listing Algorithm

- Call a cell “pure” if its count equals 1.
- While there exists a pure cell:
  - Output $x=\text{keySum}$ of the cell.
  - Call $\text{Delete}(x)$ on the IBLT.
GPU Implementation

- Each cell gets a thread.
- Each cell checks if it is pure.
  - If so, identify the key it contains and remove it from other cells in the IBLT.
  - Do this by subtracting out values in other cells.
- Issue: repeated deletion.
  - Several cells might recover and try to remove the same key in the same round. So a key gets deleted more than once!
Dealing with Repeated Deletion

- To avoid this: use $r$ subtables, such that the $i$th hash function only hashes into subtable $i$.
  - Break the listing algorithm into serial subrounds. In $i$th subround, recover only from the $i$th subtable.
  - Avoids repeated deletions, since each item will be hashed to just 1 cell in each subtable.
  - Leads to interesting variation in the analysis.

- Subrounds increase runtime, since they must happen sequentially.
  - Naively, they may blow up runtime by a factor of $r$.
  - But we show this does not happen.
    - Gains in one subround can help later subrounds.
    - We show runtime only blows up by a factor of about $\log_2(r-1)$.

- Analysis is similar to Vöcking’s $d$-left scheme.
  - Fibonacci numbers show up!
Subround Result

**Theorem B.1.** Let \( r \geq 3 \) and \( k \geq 2 \). Let \( \phi_{r-1} = \lim_{k \to \infty} F_{r-1}^{1/k}(k) \) be the asymptotic growth rate for the Fibonacci sequence of order \( r - 1 \). Let \( G \) be a hypergraph over \( n \) nodes with \( cn \) edges generated according to the following random process. The vertices of \( G \) are partitioned into \( r \) subsets of equal size, and the edges are generated at random subject to the constraint that each edge contains exactly one vertex from each set.

With probability \( 1 - o(1) \), the peeling process for the \( k \)-core in \( G \) that uses \( r \) subrounds in each round terminates after

\[
\frac{1}{r \log \phi_{r-1} + \log(k-1)} \log \log n + O(1) \text{ rounds when } c < c_{k,r}^*.
\]

**Summary:** use of \( r \) subtables increase constant factor in front of the \( \log \log n \), but by much less than a factor or \( r \).
Conclusion

- Peeling gives simple, fast greedy algorithms.
  - Usually linear or quasi-linear total work.
- Particularly well suited for parallelization.
  - Especially when aiming for an empty $k$-core.
- Implementation leads to interesting variation in the analysis.
  - Subrounds.
- Can analyze dependence on “gap” to the threshold.
Thank you!
Example 1: LDPC Codes for Erasure Channels

c_1c_2c_3c_4c_5c_6c_7c_8c_9c_{10}c_{11}c_{12}c_{13} → Erasure Channel → c_1?c_3c_4c_5?c_7?c_9c_{10}c_{11}c_{12}?
Example 1: LDPC Codes for Erasure Channels

How does an LDPC code encode an 8-bit message $m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8$?
Example 1: LDPC Codes for Erasure Channels

How does an LDPC code encode an 8-bit message \( m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8 \)?

\[ r_1 = \text{XOR}(m_1, m_3, m_5) \]
\[ r_2 = \text{XOR}(m_2, m_3, m_6) \]
\[ r_3 = \text{XOR}(m_1, m_6, m_8) \]
\[ r_4 = \text{XOR}(m_2, m_5, m_7) \]
\[ r_5 = \text{XOR}(m_4, m_7, m_8) \]
Example 1: LDPC Codes for Erasure Channels
Example 1: LDPC Codes for Erasure Channels

Decoding Algorithm:
While there exists an un-erased a parity-check bit with exactly one un-erased neighbor:
Recover the neighbor
Example 1: LDPC Codes for Erasure Channels

Decoding Algorithm:
While there exists an un-erased a parity-check bit with exactly one un-erased neighbor:
Recover the neighbor
Example 1: LDPC Codes for Erasure Channels

Decoding Algorithm:
While there exists an un-erased a parity-check bit with exactly one un-erased neighbor:
Recove the neighbor
Example 1: LDPC Codes for Erasure Channels

Decoding Algorithm:
While there exists an un-erased a parity-check bit with exactly one un-erased neighbor:
Recover the neighbor
Decoding Algorithm:
While there exists an un-erased a parity-check bit with exactly one un-erased neighbor:
Recover the neighbor
Example 1: LDPC Codes for Erasure Channels

Decoding Algorithm:
While there exists an un-erased a parity-check bit with exactly one un-erased neighbor:
Recover the neighbor
Example 1: LDPC Codes for Erasure Channels

Decoding Algorithm:
While there exists an un-erased a parity-check bit with exactly one un-erased neighbor:
Recover the neighbor
Example 1: LDPC Codes for Erasure Channels

Decoding $\leftrightarrow$ peeling to 2-core on the hypergraph $G$ where:
- Parity-check bits $\leftrightarrow$ vertices of $G$,
- Erased message bits $\leftrightarrow$ hyperedges of $G$.
- Yields capacity-achieving codes with linear encoding and decoding time [Luby, Mitzenmacher, Shokrollahi, Spielman]