Experimental Design of Work Chunking for Graph Algorithms on High Bandwidth Memory Architectures

George M. Slota$^1$ Sivasankaran Rajamanickam$^2$

$^1$Rensselaer Polytechnic Institute, $^2$Sandia National Labs
slotag@rpi.edu, srajama@sandia.gov

IPDPS 24 May 2018
Intro and Overview of Talk

- **Ongoing trend**: expansion of memory hierarchy for increased CPU throughput
  - E.g., high-bandwidth memory (HBM) layer on current generation Intel Xeon Phis (Knight’s Landing)
- Can we explicitly design graph computations to effectively utilize this layer?
- We explore a work chunking approach that iteratively brings in pieces of a large graph to perform local updates in HBM – we specifically look at the *label propagation* algorithm. We find:
  - **Chunking has minimal impact on solution quality**
  - **Chunking can also decrease time to solution**

*Primary assumption*: the graphs being processed are too large to fit entirely within MCDRAM
Intel Knight’s Landing (KNL)
68-72 cores with High Bandwidth Multi-channel DRAM (MCDRAM)

Stream Triad Bandwidth (Capacity)
- **DDR4**: 90 GB/s (up to 384 GB)
- **MCDRAM**: 450 GB/s (16 GB)

Multiple MCDRAM modes
- **Cache Mode**
- **Flat Mode**
- **Hybrid Mode**

Latency: MCDRAM \(\approx\) DDR4
Label Propagation

- Randomly label with $n = \#\text{verts labels}$
Label Propagation

- Randomly label with \( n = \text{#verts labels} \)
Label Propagation

- Randomly label with $n = \#\text{verts}$ labels
- Iteratively update each $v \in V(G)$ with max per-label count over neighbors with ties broken randomly
Label Propagation

- Randomly label with $n = \#verts$ labels
- Iteratively update each $v \in V(G)$ with max per-label count over neighbors with ties broken randomly
Label Propagation

- Randomly label with $n = \#\text{verts}$ labels
- Iteratively update each $v \in V(G)$ with max per-label count over neighbors with ties broken randomly
- Algorithm completes when no new updates possible; in large graphs, fixed iteration count
Why Label Propagation?

- **Iterative vertex updates** – prototypical of many other graph computations
- **Wide usage** – community detection, partitioning, other unsupervised learning problems
- **Nondeterministic algorithm by design** – solution quality can vary based on processing methodology
- **Suitably complex** – longer execution times might benefit from chunking optimizations
- **Straightforward to implement via work chunking**
Multilevel Memory Label Propagation via work chunking

1: \( L \leftarrow \text{LPChunking}(G(V, E), C_{\text{num}}, C_{\text{iter}}) \)
2: for all \( v \in V : L(v) \leftarrow \text{id}(v) \quad \triangleright \text{Initialize labels as vertex ids} \\
3: \text{while at least one } L(v) \text{ updates do} \\
4: \quad \text{for } c = 1 \ldots C_{\text{num}} \text{ do} \\
5: \quad \quad V_c \leftarrow \text{Chunk}(c, V), E_c \leftarrow \langle v, u \rangle \in E : v \text{ or } u \in V_c \\
6: \quad \text{for iter} = 1 \ldots C_{\text{iter}} \text{ while one } L(v) : v \in V_c \text{ updates do} \\
7: \quad \quad \text{for all } v \in V_c \text{ do in parallel} \quad \triangleright \text{Random order} \\
8: \quad \quad \quad \text{Counts} \leftarrow \emptyset \quad \triangleright \text{Hash table} \\
9: \quad \quad \text{for all } \langle v, u \rangle \in E_c \text{ do} \\
10: \quad \quad \quad \text{Counts}(L(u)) \leftarrow \text{Counts}(L(u)) + 1 \\
11: \quad \quad \text{NewLabel} \leftarrow \text{GetKeyOfMaxVal(Counts(\ldots))} \\
12: \quad \quad \text{if NewLabel} \neq L(v) \text{ then} \\
13: \quad \quad \quad L(v) \leftarrow \text{NewLabel}
Chunking Considerations

Primary chunking variables
- Number of total chunks ($C_{num}$)
- Work iterations performed on each chunk ($C_{iter}$)

How to determine data per chunk?
- Block methods (vertex block, edge block)
- Randomization or hashing
- Explicit partitioning

How to transfer chunked data?
- All threads transfer, then all threads work
- Overlap transfer of $c_{i+1}$ with work on $c_i$
- Vary number of work/transfer threads to ensure balance
Algorithmic Variants

Baseline Cache
- Baseline implementation running in cache mode

Baseline Hybrid
- Baseline implementation with hash table allocated in MCDRAM
- Graph structure and other data handled by MCDRAM cache

Chunk-HBM
- All data explicitly allocated in MCDRAM
- Per-chunk graph structure transferred into MCDRAM
- All vertex labels static in MCDRAM
**Test System:** Bowman at Sandia Labs – each node has a KNL with 68 cores, 96 GB DDR, and 16 GB MCDRAM

**Test Graphs:**

<table>
<thead>
<tr>
<th>Network</th>
<th>(n)</th>
<th>(m)</th>
<th>(d_{avg})</th>
<th>(d_{max})</th>
<th>(\tilde{D})</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>4.8 M</td>
<td>69 M</td>
<td>18</td>
<td>20 K</td>
<td>18</td>
</tr>
<tr>
<td>Friendster</td>
<td>66 M</td>
<td>1.8 B</td>
<td>27</td>
<td>5.2 K</td>
<td>34</td>
</tr>
<tr>
<td>Twitter</td>
<td>52 M</td>
<td>2.0 B</td>
<td>37</td>
<td>3.7 M</td>
<td>19</td>
</tr>
<tr>
<td>Host</td>
<td>89 M</td>
<td>2.0 B</td>
<td>22</td>
<td>3.4 M</td>
<td>23</td>
</tr>
<tr>
<td>uk-2007</td>
<td>105 M</td>
<td>3.3 B</td>
<td>31</td>
<td>975 K</td>
<td>82</td>
</tr>
<tr>
<td>wBTER_50</td>
<td>50 M</td>
<td>1.2 B</td>
<td>24</td>
<td>110 K</td>
<td>12</td>
</tr>
<tr>
<td>wBTER_100</td>
<td>100 M</td>
<td>2.4 B</td>
<td>24</td>
<td>135 K</td>
<td>12</td>
</tr>
</tbody>
</table>
How does chunking impact solution quality?
Convergence and Solution Quality
For label propagation and community detection algorithms in general

Defining convergence

- **True convergence**: no more label updates can occur
- **Looser criteria**: fixed iterations, some modularity gain or change, number of labels, others

We run to true convergence when possible, but fix iterations to enable a parametric study of chunking variables.

Defining solution quality

- Standard metrics when no ground truth exists: *modularity*, *conductance*, among many others
- When ground truth exists: *normalized mutual information* (NMI) and related measurements

Despite some observed flaws with their usage, we select the standard measurements of *modularity* and *NMI*. 
Chunking Parameters
Evaluating impact of number of chunks and iterations per chunk

- Heatmaps of iterations to convergence (left) and impact on final modularity (right) – lighter is better
- About $5 \times$ increase in iterations captured in left plot and 2% total modularity change in right plot
- While chunking increases iterations to convergence, **it has minimal impact on final solution quality** (and actually improves it in several instances – LiveJournal, Host, wBTER)
Ran same parametric tests on LFR benchmark 
\( (n = 10,000, k = 15, maxk = 500, t1 = 2, t2 = 1, \mu = 0.05 \ldots 0.6) \)

Heatmap of iterations to convergence (left) and NMI versus baseline (right) 

Similar takeaways to real-world test instances
Can HBM chunking improve time to solution?
Effect of Partitioning Methodology
5 iterations per chunk, minimum number of chunks possible (∼5), 40 iterations

- Effects of partitioning method on per-iteration speedup vs. baseline timing (left) and modularity (right)
- Explicit partitioning demonstrates largest improvements, but at the obvious cost of computing the partition
# Overall: Cache vs. Hybrid vs. Flat modes

Best times (in seconds) in each mode for each graph for 40 iter or convergence

<table>
<thead>
<tr>
<th>Network</th>
<th>Cache</th>
<th>Hybrid</th>
<th>Flat</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>33</td>
<td>29</td>
<td>25</td>
<td>P-OL</td>
</tr>
<tr>
<td>Friendster</td>
<td>495</td>
<td>337</td>
<td>333</td>
<td>VB</td>
</tr>
<tr>
<td>Twitter</td>
<td>1,793</td>
<td>871</td>
<td>242</td>
<td>P-OL</td>
</tr>
<tr>
<td>Host</td>
<td>2,447</td>
<td>2,086</td>
<td>712</td>
<td>EB-OL</td>
</tr>
<tr>
<td>uk-2007</td>
<td>1,981</td>
<td>1,241</td>
<td>783</td>
<td>P-OL</td>
</tr>
<tr>
<td>wBTER_50</td>
<td>577</td>
<td>474</td>
<td>225</td>
<td>VB-OL</td>
</tr>
<tr>
<td>wBTER_100</td>
<td>1,602</td>
<td>491</td>
<td>435</td>
<td>EB-OL</td>
</tr>
</tbody>
</table>

Partitioning: **VB**: Vertex Block; **EB**: Edge Block; **P**: PuLP

- **OL** indicates with overlapping communication
Time and modularity vs. iterations
Per-iteration time and total time doesn’t tell the whole story

Friendster (left) and Twitter (right) for modularity vs. iterations (top) and time per iteration (bottom). Baseline and $C_{num/C_{iter}}$. 
Discussion: Generalization

To other vertex programs on KNLs with HBM
- Tested chunked versions of PageRanks and K-cores
- Speedups still there but much less – under 25%
  - Hash table for label propagation is likely just extremely ill-performant in cache mode; benefits most from memory considerations
- Minimal impact on solution quality for PR (for K-cores, we run to true convergence)

GPU and SSD-based graph processing
- Note: biggest general takeaway is running multiple local iterations doesn’t impact solution quality
- So limited-memory GPUs and large-scale processing with SSD arrays might consider similar approaches

Distributed processing
- Equivalence to only communicating every nth iteration
Conclusions and future work

- Chunking minimally affects solution quality of label propagation, but can increase the number of iterations required for a given “quality”
- Explicit handling of HBM generally improves per-iteration timing and can improve time-to-solution in select instances

Future work:
- Further explore generalizations to other vertex programs
- Multi-tiered chunking – hold key vertices in HBM and update every iteration

www.gmslota.com, slotag@rpi.edu