Triangle Counting Via Vectorized Set Intersection

Shahir Mowlaei
Department of Neurological Surgery
University of Pittsburgh
Pittsburgh, Pennsylvania
Email: shahir@vt.edu

***Abstract*—In this paper we propose a vectorized sorted set intersection approach for the task of counting the exact number of triangles of a graph on CPU cores. The computation is factorized into reordering and counting kernels where the reordering kernel builds upon the Reverse Cuthill-McKee heuristic.**

I. Introduction

Triangles are among the most basic, yet nontrivial, structural units of information in the graph space. A knowledge of the set of triangles of a graph can be readily utilized towards higher level statistical and topological inference schemes of which the Clustering Coefficient [1] is a prevalent example. Due to the computational cost of structural graph queries, considerable effort is invested in improving the performance of such measures. In this work, and in contribution to the Static Graph Challenge [2], we present a vectorized set intersection approach (prefaced by a reordering step) to the problem of counting the exact number of triangles of a graph on CPU cores.

Perhaps due to its minimal structural content, a wide array of sequential, parallel and distributed algorithms have been proposed to attack the triangle counting problem. An exhaustive account of better known sequential algorithms is presented in [3] where the authors further a number of competitive algorithms that, along with their later improvements and extensions [4]–[6], are among the fastest competitors. Most notably, the *forward* algorithm achieves high speedups compared to similar algorithms by (starting from a degree sorted configuration) dynamically delegating the task of matching the neighborhoods of lower-degree vertices to their higher-degree neighbors and thereby improving data (hence cache) locality. This dynamic load transfer is, in essence, similar to the Reverse Cuthill-McKee (RCM) algorithm from finite element analysis that is heuristically tailored towards profile reduction of sparse matrices. Inspired by this connection, in this work we utilize the Reverse Cuthill-McKee algorithm to facilitate the task of counting the number of triangles in large graphs via sorted set intersection in vector registers.

II. Algorithm

The *bandwidth* of a (sparse) symmetric or triangular matrix is defined as the maximal row-wise distance of nonzero elements from its diagonal and bandwidth (and/or profile) reduction algorithms [7]–[11] attempt to optimize such measures to facilitate subsequent matrix operations such as Gaussian elimination. This class of algorithms bears the prospect of bringing the nonzero elements of a symmetric matrix closer together around the diagonal and belong to the larger class of graph layout problems [12]. On the other hand, what the *forward* algorithm of Schank et al. [3] dynamically achieves, in terms of the adjacency matrix of a graph, is to settle as many nonzero elements from the top of upper triangular sector at the bottom of its lower triangular counterpart while avoiding information loss. This allows for restriction of the neighbor matching computations to within a smaller set of vertices that, in turn, increases the efficiency of memory access patterns; similar approaches can be found in the literature [**?**], [13], [14].

The set intersection strategy that we adopt in this paper, attempts to adhere to this mentality but with a lower memory footprint in the counting phase at the expense of additional reordering costs. If it happens that similar graph algorithms can benefit from the induced data locality that is achieved at the reodering stage, our move will be better justified; but we show that, for the particular counting strategy that is employed here, and for the set of graphs on which the performance of our kernels are examined, the combination reordering and counting kernels achieves better runtimes compared to those of the counting kernel alone. To this end, in the counting kernel, we restrict our queries to the upper triangular sector of the adjacency matrix (a choice that inhibits the possibility of neighborhood updates). Since we relocate the computational cost of achieving what dynamic neighborhood updates could deliver to the reordering kernel, the competitive performance gain in the counting kernel is, by account of the obtained data *locality*, potentially recyclable for similar neighbor matching graph queries.

What is meant by locality here is a notion similar to profile minimization in the context of sparse matrix reordering. Consider the following objective function on graph $G[V, E]$,

$$f\left(G[V,E;\mathcal{I}]\right) = \sum_{v\in G}\sum_{v'\in N_+(v)} \frac{|N_+(v)\cap N_+(v')|}{M\Big(\left[N_+(v)\right],\left[N_+(v')\right]\Big)} \quad , \qquad (1)$$

where $N_+(\cdot)$ are forward neighborhoods that are induced by the ordering $\mathcal{I}$ of V,

$$N_+(v) \equiv \{v' \in G \,:\, (v,v') \in E \,\wedge\, v' \succ_{\mathcal{I}} v\} \quad , \qquad (2)$$

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

and vertices are identified with their index in the ordering notation. Finally, $M(\cdot,\cdot)$ is a generic averaging function on the *width* of forward neighborhoods $[N_+(\cdot)]$, that is, the difference between the smallest and largest assigned indices of their elements. However, in contrast to bandwidth-like objective functions that are pertinent to sparse matrix reordering, the dependence on the distance from diagonal has been relaxed in favor of a measure of neighborhood overlaps that is also sensitive to their *density*, that is, the ratio of cardinality to width of the set. While we do not manifestly optimize an objective function in class $[f]$, the formulation frames the basis on which our reordering kernel operates: we attempt to heuristically maximize such objective functions in order to increase the overlap of linked forward neighborhoods while moderating their size for the sake of cache affinity.

To arrive at an reordering that encourages the larger objective functions of class $[f]$, we adopt the Reverse Cuthill-Mckee algorithm but also reverse the direction of (degree) comparator; in other words, we initially prioritize higher degree vertices and their neighbors, but at the end, reverse the ordering. The motivation (or rather expectation) behind this heuristic, as we previously stated, is to obtain, for higher degree vertices, forward neighborhoods that strike a balance between their width and density, but also maintain a moderate relative width to avoid frequent cache misses on larger sparse graphs.

While the original RCM algorithm attempts to relocate the nonzero elements (edges) around the main diagonal, the reversion of degree comparator, allows us to lead the overall edge pattern towards the right (and bottom) of the adjacency matrix. This also differs from the strategy adopted by the forward algorithm [3] in that our inquiry, and hence our access pattern, is confined only to the upper triangular sector. The degree to which edges are pushed to the right (and bottom) is, in part, influenced by clustering pattern of the underlying graph; but a reordering that is optimizing the class $[f]$ should, generally speaking, push the edge pattern towards the position (index) of higher degree vertices of different clusters. The

Data: $G\,[V, E]$
Result: T (number of triangles in G)
$T \leftarrow 0$;
for $v \in V$ **do**
  for $v' \in N_+(v)$ **do**
    $T \leftarrow T + |N_+(v) \cap N_+(v')|$
  end
end

Algorithm 1: Counting Kernel

counting kernel, then adopts a vectorized sorted set intersection [16]–[19] strategy in anticipation of the neighborhood pattern that the reordering kernel will output. Therefore, the reordered forward neighborhoods are further sorted before passing the data to this kernel. The kernel simply iterates over the vertices and for each (active) vertex, intersects its forward neighborhood with that of its neighbors by taking advantage of vectorized loads of neighborhoods of this vertex and those of its neighbors; Algorithm 1 summarizes this approach. The induced ordering can be further utilized for dynamically reducing the forward neighborhood of the active vertex while traversing the (upper) triangular sector, since, by construction,

$$\forall\, v' \succ_{\mathcal{I}} v \;:\; v \notin N_+(v')\;; \tag{3}$$

in particular, the intersection of the active vertex and its last neighbor is always empty —an observation, that becomes relevant in vectorized implementations.

III. Implementation

We implemented Serial and parallel shared-memory realizations of this approach. Our starting point is the adjacency lists provided in the Matrix Market [20] coordinate format of undirected graphs.

A. Memory Layout

The loader is a small C routine that reads the provided Matrix Market format into full (as opposed to forward) neighborhood arrays. Two auxiliary arrays, one containing the size of neighborhoods of each vertex and the other housing the pointers to the above memory areas complement the memory landscape. During the course of entire implementation we maintain a space requirement of at most $2 \cdot E + (3 + c) \cdot V$ in multiples of 4 bytes where $c \leq 7$.

This memory layout is then passed to the reordering kernel during which a task-specific implementation of the RCM algorithm permutes the content of the above containers. However, before passing these array to the counting kernel, the actual neighborhoods are aligned on 32 byte boundaries and further padded so that the size of each neighborhood is a multiple of 32 bytes (8 neighbors). The choice of 32 simply reflects the fact that our implementation targets the Intel AVX2 technology. This alignment allows us to take advantage of presumably faster aligned loads. The padding elements are filled with the sum of the index of the corresponding vertex and the total number of vertices. This assignment is chosen to provide distinct padding elements for distinct neighborhoods which becomes relevant in vector set intersection; therefore our implementations, as it stands, can only accommodate graphs with at most 2,147,483,647 nodes as we store the indices in 4-byte integers. It is possible to assign the padding indices during intersection in the counting kernel and reclaim the last bit, but we do not entertain this entertain this possibility here. The total memory cost of the counting kernel is, therefore, $2{\cdot}E{+}\big(3{+}P(V,E,I)\big){\cdot}|V|{+}r$ where $r \leq V$. The *padding cost* $P(V, E, B_e)$ is further dependent on, B_e, the effective bitness of the implemented vectorization which, for AVX2 with 4-byte vertex indices, amounts to $256/32$; it holds that

$$0 \leq P(V, E, B_e) \leq B_e - 1 \quad ; \tag{4}$$

this coincides with the overall memory cost that we portrayed earlier.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

B. Counting Kernel

For the counting kernel, we choose to confine ourselves to the register space and limit our memory interactions to loads only. For this reason, the counting kernel was written in X86 Assembly for Flat Assembler which allows us to conveniently manipulate all general purpose registers —in addition to MMX and YMM registers.

Our sorted set intersection strategy borrows from a vast literature on this topic but since we rely on the reordering kernel to provide some level of data locality, and since we have a limited number of registers at our disposal, we suffice to comparing the first and last elements of the two vector registers in order to assess the possibility of nonempty intersections. However, as we mentioned earlier, the directed acyclic structure of our reduced problem allows us to progressively abandon portions of the active (vertex) neighborhood based on the index of visited neighbors.

Our parallel implementation of the counting kernel is based on a simple static scheduling in which each thread starts from a small index (of the active vertex) and iterates by a common step; as such it is conceivable that dynamic scheduling could improve the parallel performance of this kernel.

C. Reordering Kernel

Our reordering kernel (written in C) is a simplified version of the RCM algorithm that is tailored towards our particular data structure; however, the algorithm has been implemented in various software packages that target either sparse matrix or graph data structures [21].

Our parallel implementation of the reordering kernel suffers from dispensable serializations; the initial degree-based sort of all vertices, as well as the Breadth-First Search (BFS) in the RCM algorithm, can benefit from parallel implementations that were ignored in our current code.

IV. Experimental Results

The experiments were carried out on a single, as well as 28 cores of a dedicated cluster node [22], [23] of 2 Intel Haswell (E5-2695 v3) CPUs (2.3–3.3 GHz) with 14 cores. Table IV details the graph benchmarks that were examined along with the number of reported triangles in this implementation. The first and second groups of (synthetic) graphs, G_1 and G_2, belong to different sets of Graph 500 [24] benchmarks and are indexed by their scale. The G_1 graphs, as well the Friendster network [25], were provided as part of the Static Graph Challenge [2], while the G_2 set was first made available by the 10th DIMACS Implementation Challenge and was obtained from the University of Florida Sparse Matrix Collection [26].

In Table IV we report the median runtime of reordering and counting kernels per graph benchmark for 8 runs of the serial implementation. The time spent in forming the forward neighborhoods is included in the runtime of reordering kernel; this step involves an index-based sorting routine per neighborhood —after permutation of indices. We note that a relatively small fraction of the compute time is spent in the reordering kernel since the RCM algorithm of reordering kernel is essentially an augmented BFS; a sizable portion of reordering time is devoted to (degree-based) sort routines. Table IV summarizes the results for out parallel implementation on 28 CPU cores; we note that the ordering kernel, due to serial implementations of its (degree-based) sort and RCM routines, does not properly scale. Comparison with results of Table IV for counting without reordering, we observe that reordering has a positive impact on the performance of the counting kernel as well as the total time. In case of G_3, we attribute the absence of performance gain in the counting kernel to proper ordering of vertices (based on network topology) in the original dataset; when such knowledge is available, either through external agents or by means of heuristics, the reordering kernel can be avoided in favor of reduction of the total compute time.

TABLE I
Examined graph benchmarks and their obtained triangle count.

Graph	Vertices	Edges	Triangles
G_1^{18}	174,147	7,600,696	82,287,285
G_1^{19}	335,318	15,459,350	186,288,972
G_1^{20}	645,820	31,361,722	419,349,784
G_1^{21}	1,243,072	63,463,300	935,100,883
G_1^{22}	2,393,285	64,097,004	2,067,392,370
G_1^{23}	4,606,314	129,250,705	4,549,133,002
G_1^{24}	8,860,450	260,261,843	9,936,161,560
G_1^{25}	17,043,780	523,467,448	21,575,375,802
G_2^{16}	65,536	2,456,398	118,811,321
G_2^{17}	131,072	5,114,375	287,593,439
G_2^{18}	262,144	10,583,222	687,677,667
G_2^{19}	524,288	21,781,478	1,625,559,121
G_2^{20}	1,048,576	44,620,272	3,803,609,518
G_2^{21}	2,097,152	91,042,010	8,815,649,682
G_3	65,608,366	1,806,067,135	4,173,724,142

TABLE II
Median of serial runtimes (in seconds, of 8 runs) of reordering and counting kernels and their sum per run.

Graph	Reordering	Counting	Total Time
G_1^{18}	0.311	0.871	1.168
G_1^{19}	0.692	2.289	2.978
G_1^{20}	1.497	5.765	7.236
G_1^{21}	3.135	17.146	20.289
G_1^{22}	6.593	47.010	53.576
G_1^{23}	14.423	122.734	137.221
G_1^{24}	31.794	306.838	338.637
G_1^{25}	68.841	753.805	822.590
G_2^{16}	0.194	0.655	0.842
G_2^{17}	0.410	1.797	2.177
G_2^{18}	0.893	4.797	5.646
G_2^{19}	2.069	13.793	15.837
G_2^{20}	4.403	38.953	43.346
G_2^{21}	9.059	102.485	111.555
G_3	279.494	1356.001	1636.732

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE III
MEDIAN OF PARALLEL (28 CORES) RUNTIMES (IN SECONDS, OF 16 RUNS) OF REORDERING AND COUNTING KERNELS AND THEIR SUM PER RUN. NUMBERS IN PARENTHESES STAND FOR SPEEDUPS OVER THE SERIAL AND WITHOUT-REORDERING IMPLEMENTATIONS, RESPECTIVELY.

Graph	Reordering	Counting	Total
G_1^{18}	0.109 (2.9)	0.053 (16.4, 5.9)	0.166 (7.0, 2.2)
G_1^{19}	0.214 (3.2)	0.092 (24.9, 8.4)	0.307 (11.3, 2.8)
G_1^{20}	0.412 (3.6)	0.211 (27.3, 9.9)	0.640 (11.3, 3.4)
G_1^{21}	0.851 (3.7)	0.571 (30.0, 8.9)	1.430 (14.2, 3.7)
G_1^{22}	1.843 (3.6)	1.528 (30.8, 8.9)	3.394 (15.8, 4.1)
G_1^{23}	3.965 (3.6)	4.227 (29.0, 8.7)	8.216 (16.7, 4.5)
G_1^{24}	8.408 (3.8)	10.729 (28.6, 9.1)	19.300 (17.5, 5.1)
G_1^{25}	17.769 (3.9)	26.863 (28.1, 8.9)	44.701 (18.4, 5.4)
G_2^{16}	0.079 (2.5)	0.041 (16.0, 3.0)	0.122 (6.9, 1.3)
G_2^{17}	0.115 (3.6)	0.086 (20.9, 3.2)	0.203 (10.4, 1.6)
G_2^{18}	0.213 (4.2)	0.183 (26.2, 4.5)	0.395 (14.3, 2.3)
G_2^{19}	0.438 (4.7)	0.484 (28.5, 4.4)	0.915 (17.3, 2.5)
G_2^{20}	0.915 (4.8)	1.321 (29.5, 4.4)	2.243 (19.3, 2.6)
G_2^{21}	2.013 (4.5)	3.696 (27.7, 3.9)	5.696 (19.6, 2.6)
G_3	76.885 (3.6)	45.230 (30.0, 1.0)	122.001 (13.4, 0.4)

TABLE IV
MEDIAN OF PARALLEL (28 CORES) RUNTIMES (IN SECONDS, OF 16 RUNS) OF REORDERING* AND COUNTING KERNELS AND THEIR SUM PER RUN. HERE, THE REORDERING KERNEL ONLY FORMS THE FORWARD NEIGHBORHOODS WITHOUT PERMUTING THE INDICES.

Graph	Reordering*	Counting	Total Time
G_1^{18}	0.049	0.315	0.363
G_1^{19}	0.070	0.774	0.845
G_1^{20}	0.084	2.095	2.178
G_1^{21}	0.150	5.090	5.252
G_1^{22}	0.231	13.524	13.755
G_1^{23}	0.358	36.573	36.936
G_1^{24}	0.738	98.124	98.864
G_1^{25}	1.468	240.037	241.460
G_2^{16}	0.040	0.124	0.163
G_2^{17}	0.053	0.279	0.332
G_2^{18}	0.072	0.819	0.890
G_2^{19}	0.119	2.131	2.256
G_2^{20}	0.175	5.762	5.931
G_2^{21}	0.308	14.425	14.729
G_3	2.718	44.552	47.366

V. CONCLUSION

We proposed the adoption of a well-known and practiced BFS, the Reverse Cuthill-McKee algorithm, for reordering the vertices as a pre-processing step to the triangle counting problem. Our choice of sorted set intersection for the counting kernel is aligned with what we expect from the output of the reordering kernel and, inspired by similar work on sorted set intersection, takes advantage of vector registers towards this end. While our implementation can benefit from careful optimization, the benchmark results are promising and, with the arrival of wider registers and more flexible instructions, higher performance gains are conceivable.

ACKNOWLEDGMENT

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

REFERENCES

[1] D. J. Watts and S. H. Strogatz, *Collective dynamics of small-world networks*, Nature, 393(6684):440442, 1998.
[2] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli and J. Kepner, *Subgraph Isomorphism*, IEEE HPEC, 2017. https://graphchallenge.mit.edu/challenges
[3] T. Schank and D. Wagner, *Finding, counting and listing all triangles in large graphs, an experimental study*, in *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, ser. WEA05. Berlin, Heidelberg: Springer-Verlag, 2005.
[4] Schank, Thomas. *Algorithmic aspects of triangle-based network analysis*, 2007.
[5] M. Latapy, *Main-memory triangle computations for very large (sparse (power-law)) graphs*, Theor. Comput. Sci., vol. 407, no. 1-3, Nov. 2008.
[6] S. Suri and S. Vassilvitskii, *Counting triangles and the curse of the last reducer*, in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW 11. New York, NY, USA: ACM, 2011.
[7] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in *Proceedings of the 1969 24th national conference*, pages 157-172, New York, NY, USA: ACM, 1969.
[8] W. Liu and A. H. Sherman, *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices*, SIAM Journal on Numerical Analysis, 13(2):198-213, 1976.
[9] N. E. Gibbs, W. G. Poole, and P. K. Stockemeyer, *An algorithm for reducing the bandwidth and profile of a sparse matrix*, SIAM Journal of Numerical Analysis, 13(2):236-250, April 1976.
[10] A. George and J. W. H. Liu, *An implementation of a pseudoperipheral node finder*, ACM Trans. Math. Softw., 5(3):284-295, 1979.
[11] A. George and J. W. H. Liu *Computer solution of large sparse positive definite systems*, Prentice-Hall series in computational mathematics. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.
[12] J. Díaz, J Pettit and M. Serna, *A survey of graph layout problems*, ACM Comput. Surveys, 34, pp. 313-356, 2002.
[13] S. Arifuzzaman, M. Khan, and M. Marathe, *PATRIC: A parallel algorithm for counting triangles in massive networks*, in *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM 13, pp. 529538, New York, NY, USA: ACM, 2013.
[14] M. Ortmann and U. Brandes, *Triangle listing algorithms: back from the diversion*, in *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments* (ALENEX), 2014.
[15] G. M. Slota, S. Rajamanickam and K. Madduri, *Order or shuffle: empirically evaluating vertex order impact on parallel graph computations*, in *Proceedings of Parallel and Distributed Processing Symposium Workshops*, (IPDPSW), IEEE International, 2017.
[16] B. Schlegel, T. Willhalm and W. Lehner, *Fast Sorted-Set Intersection using SIMD Instructions*, ADMS@VLDB, 2011.
[17] H. Inoue, M. Ohara and K. Taura, *Faster set intersection with SIMD instructions by reducing branch mispredictions*, Proc. VLDB Endow. 8, 3 (November 2014), 293-304, 2014.
[18] D. Lemire and L. Boytsov, *Decoding billions of integers per second through vectorization*, Softw. Pract. Exper., 45, 129, 2014.
[19] D. Lemire, L. Boytsov and N. Kurz, *SIMD compression and the intersection of sorted integers*, Softw. Pract. Exper., 46: 723749, 2016.
[20] Matrix Market Repository. http://math.nist.gov/MatrixMarket/
[21] Intel Labs, *Sparse Matrix Pre-processing Library*, 2015. https://github.com/IntelLabs/SpMP
[22] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott and N. Wilkens-Diehr, *XSEDE: Accelerating Scientific Discovery*, Computing in Science & Engineering. 16(5):62-74, 2014.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

[23] N. A. Nystrom, M. J. Levine, R. Z. Roskies and J. R. Scott,*Bridges: A Uniquely Flexible HPC Resource for New Communities and Data Analytics*, In Proceedings of the 2015 Annual Conference on Extreme Science and Engineering Discovery Environment (St. Louis, MO, July 26-30, 2015). XSEDE15. ACM, New York, NY, USA, 2015.

[24] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang., *Introducing the graph 500*, Cray Users Group (CUG), 2010.

[25] J. Yang and J. Leskovec, *Defining and Evaluating Network Communities based on Ground-truth*, ICDM, 2012.

[26] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, NA Digest Vol. 92, No. 42, Oct. 23, 1994. `http://www.cise.ufl.edu/research/sparse/matrices`

978-1-5386-3472-1/17/$31.00 ©2017 IEEE