Hierarchical Sparse Graph Computations on Multicore Platforms

Humayun Kabir Kamesh Madduri
Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
Email: {hzk134, madduri}@cse.psu.edu

Accelerating sparse graph computations is challenging. This is because the degree distribution of the vertices is skewed. The time needed to process a vertex is proportional to the number of adjacencies and thus processing time of the vertices is not uniform. Additionally, the memory access pattern of the sparse graph algorithms is very irregular. Also, the modern processor has complex memory systems with multiple levels of caches. Thus shared memory parallel algorithms for sparse graph computations need to consider this complex memory system and need to keep the threads busy by dividing the work among the threads equally.

To understand the structure of graphs, it is often useful to find the densely connected or cohesive subgraphs in a graph. In the literature, there are many definition of cohesive subgraphs. Among them, maximal cliques, n-clique, n-clan [1] and n-club [1] are well known. However, all of these cohesive subgraph problems are NP-complete and thus they are computationally expensive.

The cohesive subgraphs defined using k-core [2], [3] and k-truss [4] have hierarchical structures and can be computed exactly in polynomial time. A k-core is defined as a subgraph such that each vertex in the subgraph has degree at least k. A k-truss is defined as a subgraph such that each edge in the subgraph is contained in at least $k-2$ triangles. A vertex has coreness value k if it belongs to k core, but not to $k+1$-core. Similarly, an edge has trussness value k if it belongs to k-truss but not to $k+1$-truss. k-core decomposition refers to finding the coreness value of each vertex in the graph. Similarly, k-truss decomposition refers to finding the trussness value of each edge in the graph. k-core and k-truss decompositions have many uses in large-scale graph analysis, including visualization [5], preprocessing for community detection [6] and maximal clique finding [7]. k-truss decomposition is also included in the HPEC 2017 static graph challenge [8].

Sparse matrix kernels are useful in solving system of linear equations, scientific computing and engineering. Sparse matrix kernels, such as, sparse-matrix vector multiplication (SpMV) and sparse triangular solution (STS) takes the majority of time when solving a system of linear equations using iterative method. These kernels are also challenging to accelerate on shared memory systems, because of low ratio of floating operations to memory access and irregular memory access pattern.

In my research, I develop parallel algorithms on shared memory systems for efficiently computing hierarchical cohesive subgraphs in a graph. I also develop parallel algorithms for accelerating sparse matrix kernels.

1. Methodologies

In this section, we present our algorithms for hierarchical cohesive subgraph computation and speeding up sparse matrix kernels.

PKC: parallel k-core decomposition Our algorithm PKC [9], for k-core decomposition, works in a level by level fashion. It finds all the vertices belonging to l-core, before finding vertices belonging to $(l+1)$-core. In PKC, an array deg is used to store the coreness value of the vertices and initially it contains the degree of each vertex, which is the upper bound of coreness value. To find vertices belonging to l-core in parallel, the deg array is scanned and the threads add vertices with degree l to its local buffer $buff$. Each thread processes the vertices in $buff$ until it becomes empty. This may add new vertices to l-core and these vertices are added to $buff$. When $buff$ becomes empty, l is incremented and vertices belonging to $(l+1)$-core is found. The algorithm terminates when no vertices are present in deg with degree l. If $c_{\max}$ denotes the maximum k-core of a graph, the time complexity of PKC is $O(c_{\max} * n + m)$, where n and m are number of vertices and edges respectively.

PKT: parallel k-truss decomposition The outline of our algorithm PKT [10], [11] for k-truss decomposition is given in Algorithm 1. PKT starts by computing the support of each edge in parallel and stores the support in S. For support computation we use the triangle counting algorithm from [12], that puts a direction on the edges and counts a triangle exactly once. PKT processes the edges similar to PKC in a level by level fashion. To find the edges with trussness value l, the threads scan the array $S$and add the edges with support equal to l to the $curr$ array. The edges in $curr$ are processed in a loop. An edge is processed by processing the triangles containing this edge. This decreases the support of each edge in a triangle. This adds new edges to level l and they are added to $next$. $curr$ and $next$ are swapped and processing continues, until $curr$ becomes empty. Since a triangle could be processed by more than one thread, we utilize the edge numbers of a triangle and the thread that

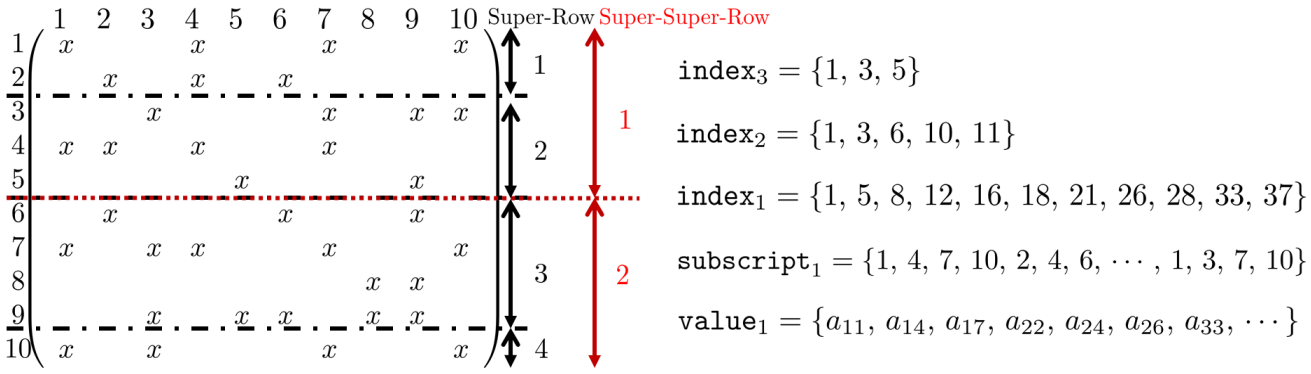

Figure 1. An illustration of *CSR-k* for $k = 3$. Level two corresponds to `index2` with super-rows shown using black-lines and level three corresponds to `index3` with super-super-rows shown using red lines.

contains the minimum number edge, processes the triangle. This enforces that a triangle is processed only once. If $t_{\max}$ denotes the maximum truss in the graph, the time complexity of PKT is: $O(t_{\max} * m + \sum_v d(v)^2) = O(\sum_v d(v)^2)$. In a graph, a wedge is defined as a pair of edges with a common end point, for example, $\langle u, v\rangle$ and $\langle v, w\rangle$. The number of wedges in a graph is $|\wedge| = (\sum_v d(v)^2 - 2m)/2$. Thus $|\wedge|$ is an estimation of the work performed by PKT.

Algorithm 1 PKT: Parallel k-truss decomposition.

```
procedure PKT(G)
    S ← φ                                   ▷ Global trussness array
    curr ← φ, next ← φ                      ▷ Temp arrays
    SUPPORTCOMP(G, S)
    todo ← m; k ← 0
    while todo > 0 do
        SCAN(S, k, curr)
        while |curr| > 0 do
            todo ← todo − |curr|
            PROCESSSUBLEVEL(S, k, curr, next)
            Swap curr and next
        k ← k + 1
```

Graph Analysis using Hierarchical Graph Computations We use hierarchical graph computations (k-core and k-truss decomposition) to analyze sparse graphs. Hierarchical graph computations can be used for graph coarsening, sparsification, vertex reordering, partitioning and community detection. We develop reordering based on SlashBurn [13]. SlashBurn tries to compress a graph using less number of bits and also decreases number of non-empty blocks in the graph. SlashBurn reorders a graph by repeatedly removing k vertices with highest centrality. In our approach, we remove all the vertices in the maximum k-core or k-truss.

***CSR-k*: A Multilevel Compressed Sparse Row Format for Sparse Matrix Computation** We have developed *CSR-k* to store a sparse matrix and compute sparse kernels using it. To store a matrix in *CSR-k* format, the graph of the matrix is considered. The graph is coarsened (k-1)-times to get a coarsened graph. Now, to decrease the bandwidth of the matrix, the coarsened graph is reordered using RCM. The reordering of the coarsened graph induces an ordering on the finer graph. Also, the coarsening is done in a way that put equal number of non-zeros in each super-rows. An example is shown in Figure 1, that represents a graph in CSR-3 format.

SpMV using *CSR-k* SpMV can be performed using the *CSR-k* representation of a matrix. For example, if *CSR-2* is used to store a matrix, the coarsest graph is used to compute SpMV . Each super-row of the coarsest graph contains a number of rows (vertices) of the input matrix. The super-rows are distributed among the threads and the threads compute the result for all the rows contained in a super-row. The details of the algorithm and the data structure can be found in [14].

STS using *CSR-k* To compute STS in parallel, the coarsened graph is colored or a level set is performed on the coarsened graph. The coarsened graph is reordered according to the size of the color, to increase reuse in unknowns. This reordering induces an ordering on the original graph. Now, the super-rows in each color are solved in parallel [15]. A color c is solved, once all the colors before c have been solved. Since, there is dependency among the colors, so a synchronization is needed after each color.

2. Performance Results and Analysis

We evaluate the methods on a dual-socket Intel shared memory server with 128 GB main memory. The shared memory node contains two 2.2 GHz Xeon E5-2650 v4 processors (Brodwell). Each processor has twelve cores and 30 MB L3 cache.

PKC performance We compare the performance of PKC [9] with ParK [16], and MPM [17] methods. The execution time of the methods on 24 threads for fifteen sparse graphs is given in Table 1. We observe that PKC outperforms both ParK and MPM for most of the graphs.

TABLE 1. EXECUTION TIME OF PKC, PARK AND MPM ON 24 THREADS.

Graph	Execution time (s)		
	PKC	ParK	MPM
cit-Patents	0.08	0.09	0.24
soc-pokec	0.10	0.08	1.03
soc-LiveJournal1	0.23	0.27	1.99
ljournal-2008	0.19	0.30	1.72
wb-edu	0.09	0.40	0.99
in-2004	0.06	0.16	0.11
as-skitter	0.06	0.07	0.23
com-orkut	0.56	0.40	4.38
hollywood-2009	0.27	0.35	1.67
uk-2002	0.39	1.17	1.02
indochina-2004	0.59	3.33	1.28
soc-friendster	10.10	7.57	113.77
webbase-2001	1.59	8.28	12.21
arabic-2005	1.33	4.42	4.80
it-2004	2.65	8.04	13.66

Among the methods, MPM takes the most amount of time for k-core decomposition. This is because MPM does a lot of redundant computations. PKC performs the best for web crawl graphs. This is because $c_{\max}$ is high for these graphs and most of the vertices are contained in first few k-core subgraphs. Since PKC switches to a smaller graph, so the scan time and processing time decrease significantly for these graphs. On 24 threads, PKC achieves a speedup (geometric mean of speedups of 15 graphs) of 1.90× as compared to ParK. A detailed evaluation including test suite and comparison to BZ [18] algorithm can be found in [9].

PKT performance The performance of PKT is reported in terms of Giga (10^9) Wedges Examined Per Second (GWeps). We report execution time, performance, relative speedup on 24 core and speedup against *Ros* in Table 2. The performance of PKT varies from 0.42 to 6.45 GWeps. The geometric mean performance for the whole test suite is 1.93 GWeps on 24 cores. The relative speedup of PKT on 24 cores varies from 6.41 to 14.29×. We compare PKT with a state-of-the-art method *Ros* [19] on 24 cores. Our method outperforms *Ros* for all the graphs and performs more than one order of magnitude better than *Ros* for most of the graphs. This is because PKT does not use hash table, it uses a data structure that is amenable to concurrent updates and is space efficient; utilizes the best method for triangle counting to compute support and it computes the trusses in parallel.

Graph analysis performance We denote the SlashBurn algorithm by SB (k) and we denote our approach by SB-core and SB-truss. We report the number of bits needed to compress each edge and the number of non-empty blocks per edge for the graphs LiveJournal and Orkut in Figure 2. We observe that our methods decrease bits need to compress a graph by more than 22% and number of non-empty blocks by 28% compared to SlashBurn.

SpMV performance using *CSR-k* The execution time of *CSR-2*, pOSKI [20] and *MKL* [21] on 24 cores is shown in Table 3. *CSR-2* performs the best among the

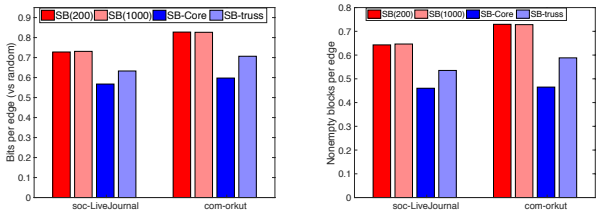

Figure 2. Number of bits needed per edge and number of non-empty blocks per edge (vs random).

TABLE 2. PKT: k-TRUSS DECOMPOSITION PARALLEL PERFORMANCE.

Graph	Time (s)	Perf (GWeps)	Speedup 24-core	Speedup over *Ros*
cit-Patents	0.8	0.42	7.86	8.51
soc-pokec	2.14	0.97	10.53	13.68
soc-LiveJournal1	7.35	0.99	9.77	12.38
ljournal-2008	9.3	1.07	10.16	12.91
wb-edu	5.77	2.11	7.63	9.42
in-2004	3.42	4.42	9.77	10.83
as-skitter	2.48	6.45	13.97	17.99
com-orkut	36.27	1.26	11.68	15.26
hollywood-2009	34.36	1.39	12.65	17
uk-2002	66.52	3.03	8.72	13.06
indochina-2004	248.15	2.3	14.29	8.97
com-friendster	1660.5	1	6.41	22.56
webbase-2001	427.8	2.89	7.88	12.06
arabic-2005	693.51	5.09	10.58	13
it-2004	3817.39	4.23	7.25	–

TABLE 3. EXECUTION TIME OF *CSR-2*, POSKI AND *MKL* ON 24 THREADS.

Matrix	Execution time (s)		
	CSR-2	pOSKI	*MKL*
ldoor	0.005	0.005	0.031
rgg_n_2_21_s0	0.004	0.005	0.025
nlpkkt160	0.027	0.031	0.107
delaunay_n23	0.008	0.017	0.037
road_central	0.008	0.025	0.036
hugetrace-00020	0.009	0.024	0.042
delaunay_n24	0.017	0.030	0.057
hugebubbles-00000	0.011	0.028	0.042
hugebubbles-00010	0.011	0.031	0.043
hugebubbles-00020	0.012	0.034	0.047
road_usa	0.014	0.038	0.036
europe_osm	0.025	0.055	0.077

methods. This is because *CSR-2* makes the super-rows with equal number of non-zeros, this helps to distribute the work equally among the threads. Also, the reordering technique helps to enhance locality in memory access pattern. pOSKI incurs zero fill-in to increase cache hit rate, however this is helpful for relatively dense matrices. For very sparse matrices this strategy does not help, because the operation count increases by zero fill-in outweighs the decrease in memory accesses. On average, *CSR-2* achieves a speedup (geometric mean of speedups of 12 matrices) of 3.81× and 2.12× compared to *MKL* and pOSKI respectively on 24 cores.

TABLE 4. EXECUTION TIME OF STS-3, COL, CSR-3-LS AND LS ON 24 THREADS.

	Execution time (s)			
Matrix	STS-3	COL	CSR-3-LS	LS
ldoor	0.006	0.007	0.007	0.009
rgg_n_2_21_s0	0.005	0.007	0.012	0.013
nlpkkt160	0.030	0.039	0.032	0.035
delaunay_n23	0.013	0.026	0.021	0.034
road_central	0.015	0.039	0.047	0.071
hugetrace-00020	0.018	0.049	0.049	0.078
delaunay_n24	0.026	0.050	0.039	0.062
hugebubbles-00000	0.020	0.057	0.072	0.101
hugebubbles-00010	0.021	0.056	0.073	0.162
hugebubbles-00020	0.023	0.061	0.068	0.113
road_usa	0.025	0.068	0.056	0.109
europe_osm	0.048	0.145	0.132	0.257

STS performance using *CSR-k* To compute sparse triangular solution (STS) using *CSR-k* , we coarsen the graph of the matrix and the coarsened graph is colored or a level-set is done on the graph. This puts the matrix in CSR-3 representation. If coloring is used on the coarsest graph, we call the method STS-3 and the method is called CSR-3-LS if level-set is used on the coarsest graph. The method COL corresponds to coloring the original graph and LS corresponds to doing a level-set on the original graph. The execution time of these methods on 24 cores is given in Table 4. We observe that STS-3 performs the best among the methods. The second best method is COL. STS-3 performs better because it forms super-rows by putting highly connected vertices tigether; this enhances locality in memory access pattern. Also the super-row sizes are equal, so work is distributed equally among the threads. Additionally, coloring exposes more parallelism compared to level-set. On average, STS-3 achieves a speedup (geometric mean of speedups of 12 matrices) of 2.21×, 2.26× and 3.49× compared to COL, CSR-3-LS and LS respectively.

3. Conclusions and Future Work

We have developed parallel algorithms to find cohesive subgraphs efficiently in large sparse graphs. Our algorithm PKC accelerate k-core decomposition by switching to a small graph when a large fraction of vertices have been processed. On 24 cores, PKC achieves a speedup of 1.90× compared to ParK. Our algorithm PKT for k-truss decomposition avoids using hash table and computes truss in parallel. PKT achieves a geometric mean performance rate of 1.93 GWEPS on a 24 core shared memory system. We have also developed a multilevel data structure *CSR-k* and we use it to speedup SpMV and STS. We would like to use PKC and PKT to analyze large sparse graphs and also like to develop graph algorithms on manycore systems (GPU, Intel Xeon Phi).

References

[1] R. J. Mokken, "Cliques, clubs and clans," *Quality & Quantity*, vol. 13, pp. 161–173, Apr. 1979.

[2] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[3] D. Matula and L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, 1983.

[4] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," National Security Agency, Tech. Rep., 2008.

[5] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Proc. Advances in Neural Information Processing Systems (NIPS)*, 2005.

[6] K. Saito, T. Yamada, and K. Kazama, "Extracting communities from complex networks by the k-dense method," in *Proc. Int'l. Workshop on Mining Complex Data (MCD)*, 2006.

[7] R. A. Rossi, D. F. Gleich, and A. H. Gebremedhin, "Parallel maximum clique algorithms with applications to network analysis," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. C589–C616, 2015.

[8] 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, "Static Graph Challenge: Subgraph Isomorphism," in *Proc. IEEE High Performance Extreme Computing (HPEC)*, 2017.

[9] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *Proc. Workshop on Parallel and Distributed Processing for Computational Social Systems (ParSocial)*, 2017.

[10] ——, "Shared-memory graph truss decomposition," arXiv.org e-Print archive, https://arxiv.org/abs/1707.02000, July 2017.

[11] ——, "Parallel k-truss decomposition on multicore systems," in *Proc. HPEC*.

[12] S. Parimalarangan, G. M. Slota, and K. Madduri, "Fast parallel graph triad census and triangle counting on shared-memory platforms," in *Proc. Workshop on Parallel and Distributed Processing for Computational Social Systems (ParSocial)*, 2017.

[13] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, vol. 26, 2014.

[14] H. Kabir, J. D. Booth, and P. Raghavan, "A multilevel compressed sparse row format for efficient sparse computations on multicore processors," in *21st International Conference on High Performance Computing (HiPC)*, 2014.

[15] H. Kabir, J. D. Booth, G. Aupy, A. Benoit, Y. Robert, and P. Raghavan, "Sts-k: A multilevel sparse triangular solution scheme for numa multicores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[16] N. S. Dasari, R. Desh, and M. Zubair, "ParK: An efficient algorithm for k-core decomposition on multicore processors," in *2014 IEEE International Conference on Big Data (Big Data)*, 2014.

[17] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," in *Proc. Symp. on Principles of Distributed Computing (PODC)*, 2011.

[18] V. Batagelj and M. Zaversnik, "An $O(m)$ algorithm for cores decomposition of networks," arXiv.org e-Print archive, http://arxiv.org/abs/cs.DS/0310049, 2003.

[19] R. A. Rossi, "Fast triangle core decomposition for mining large graphs," in *Proc. Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining (PAKDD)*, 2014.

[20] "pOSKI: Parallel Optimized Sparse Kernel Interface. bebop.cs.berkeley.edu/poski," 2012.

[21] "Intel Math Kernel Library," 2014.