ParK: An Efficient Algorithm for k-core Decomposition on Multicore Processors

Naga Shailaja Dasari, Ranjan Desh, and Zubair M
Department of Computer Science
Old Dominion University
Norfolk, USA
Email: {ndasari, dranjan, zubair}@cs.odu.edu

***Abstract*—The k-core of a graph is the largest induced subgraph with minimum degree k. The k-core decomposition is to find the core number of each vertex in a graph, which is the largest value of k that the vertex belongs to a k-core. k-core decomposition has applications in many areas including network analysis, computational biology and graph visualization. The primary reason for it being widely used is the availability of an $O(n+m)$ algorithm. The algorithm was proposed by Batagelj and Zaversnik and is considered the state-of-the-art algorithm for k-core decomposition. However, the algorithm is not suitable for parallelization and to the best of our knowledge there is no algorithm proposed for k-core decomposition on multicore processors. Also, the algorithm has not been experimentally analyzed for large graphs. Since the working set size of the algorithm is large, and the access pattern is highly random, it can be inefficient for large graphs. In this paper, we present an experimental analysis of the algorithm of Batagelj and Zaversnik and propose a new algorithm, *ParK*, that significantly reduces the working set size and minimizes the random accesses. We provide an experimental analysis of the algorithm using graphs with up to 65 million vertices and 1.8 billion edges. We compare the *ParK* algorithm with state-of-the-art algorithm and show that it is up to 6 times faster. We also provide a parallel methodology and show that the algorithm is amenable to parallelization on multicore architectures. We ran our experiments on a 4 socket Nehalem-EX processor which has 8 cores per socket and show that the algorithm scales up to 21 times using 32 cores.**

***Keywords*-k-core; parallel algorithm; multicore; graph decomposition;**

I. Introduction

The k-core of a graph is the largest induced subgraph with minimum degree k. The notion of a core was first introduced in 1983 by Seidman et al. [1]. Since then, it has been extensively studied and used in applications in many areas including network analysis, computation biology and visualization. k-core has been primarily applied in identifying the cohesive subgroups in a network. Many notions can be considered for identifying such groups including cliques, k-plexes, n-cliques[2]. While most other approaches are computationally expensive, k-core decomposition can be computed in linear time.

k-core decomposition has been used in analyzing and understanding the internet topology [3][4]. It has been used in the study of influential spreaders in complex networks [5]. It was shown that the most efficient spreaders are those located within the core of the network. It was used in detecting dense communities in large, social networks [6][7]. k-core decomposition is considered as an important tool in visualization [8][9][10]. In computational biology it was used in analyzing and detecting protein interactions [11] and analyzing gene networks [12]. k-core decomposition is used as a pre-processing step in other graph problems like find maximal and maximum cliques [13][14]. It is considered an important tool in network analysis and is included in network analysis packages[15][16][17].

The core number of a vertex is the highest value of k such that the vertex belongs to a k-core. The k-core decomposition algorithm asks to find the core number of all the vertices in the graph. It can be computed by repeatedly removing the minimum degree vertices from the graph. The main challenge in computing the k-core is to find, at each step, the minimum degree vertex in the remaining graph. This requires sorting all the remaining vertices in the graph by degree. Batagelj and Zaversnik proposed an $O(n+m)$ algorithm which is based on bin-sort. Instead of sorting at each step, the algorithm uses two arrays: one to store the vertices in sorted order of degree, and the other to store the position of each vertex in the sorted array. At each step when a vertex is removed from the graph, and the degree of its neighbors is decremented, each neighbor is moved to the appropriate position in the sorted array and its position is updated in the other array. In the rest of the paper, we use BZ to refer to the algorithm of Batagelj and Zaversnik .

BZ is considered the state-of-the-art algorithm for k-core decomposition and is widely used in many applications. However, the algorithm has not been experimentally analyzed for large graphs. In each iteration of the algorithm, i.e. for each vertex, the algorithm requires random read and write access to three different arrays of size n, where n is number of vertices in the graph. For large values of n, random access can be very expensive due to the memory latency incurred in bringing the data from main memory to cache. In this paper, we perform an experimental evaluation of the algorithm using large graphs.

Efficient implementations of linear algebra applications on multicore architectures are well understood. The cache blocking technique has been extensively applied to linear algebra applications resulting in effective utilization of mem-

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

ory bandwidth in a multicore architecture [18][19]. However, developing efficient implementation of applications dealing with large graphs is challenging, and is recently gaining attention due to the importance of solving large social network applications. The main challenge in developing efficient approaches for large graph problems is the effective utilization of caches at different levels of a multicore architecture in the presence of an unstructured access pattern. Recently, researchers have explored parallel algorithms for graph problems on multicore architectures [20][21]. In this paper, we develop an efficient scalable parallel algorithm, *ParK*, for the k-core decomposition problem on multicore architectures. To the best of our knowledge, there is no algorithm for k-core decomposition on multicore processors.

The algorithm of Batagelj and Zaversnik is less suitable for parallelization due to huge synchronization overhead. As it uses three arrays that are accessed at each step, the arrays need to be shared by multiple threads. Moreover, the values of the elements of different arrays are related, i.e. updating an element of an array requires updating corresponding elements in the other arrays (explained in section II-A). So, to avoid race conditions, more expensive synchronization constructs such as locks need to be used. There are approaches proposed for k-core decomposition of dynamic networks [22][23][24]. These approaches primarily focus on efficiently maintaining the core numbers of the vertices as the graph changes over time. This is outside the scope of this paper.

There is a distributed algorithm proposed for k-core decomposition [25]. The algorithm assigns a set of vertices to each process. For each vertex assigned to a process, it stores an array consisting of core numbers of its neighbors. The core number of all the vertices is initialized to their degree. The core number of the vertices is updated based on the core number of their neighbors. Once the core number of a vertex is updated it is communicated to the other processes to which its neighbors are assigned. This is repeated until core value of none of the vertices is updated. Each process needs to repeatedly access all the vertices assigned to it and its neighbors. This approach works fine in a distributed environment where there are large number of computing nodes and each computing node is assigned only a few vertices. This approach is not suitable for multicore architecture with only a limited number of processors as each thread is assigned large number of vertices resulting in huge working set size and hence significantly poor locality of reference.

In *ParK* algorithm, the core number of all the vertices is initialized to their degrees and the vertices are processed in levels. In each level l, all the vertices that belong to l-shell are processed (shell is described in section II) i.e. all the vertices with core number l are processed. Processing a vertex involves decrementing the core number of each of its neighbors by 1 if it has not already been processed. All the vertices processed in level l belong to the l core. Each level consists of two phases: scan phase and loop phase. In scan phase, all the vertices are scanned to collect the vertices whose core number is equal to the current level. The loop phase consists of one or more sub-levels. In the first sub-level all the vertices collected in scan phase are processed. Processing these vertices might result in additional vertices that belong to the current level. These vertices are processed in the next sub-level. This is repeated until a sub-level results in no more vertices that belong to the current level.

We note that the *ParK* algorithm shares certain similarity with the breadth first search (BFS) graph exploration. As in BFS, it proceeds in levels. Each level consists of processing some of the vertices by accessing their neighbors. In fact, the *ParK* algorithm is based on two BFS algorithms: one by Agarwal et al. [20] and the other by Hong et al. [21]. The algorithm by Agarwal et al. is a level-synchronous algorithm. It uses two queues, current queue and next queue. Current queue contains all the vertices to be processed in the current level. At the end of each level, the next queue contains all the neighbors of the vertices in the current queue that have not already been processed. The two queues are swapped at the end of each level. The algorithm by Hong et al. is a read-based method, in which at each level, all the vertices are scanned. If a vertex belongs to the current level, its neighbors are marked to be processed in the next level if they have not already been processed. The *ParK* algorithm is a hybrid algorithm that uses both the approaches. The read-based method is used in the outer level and the queue-based method is used to process the sub-levels.

We present a simple methodology to parallelize the *ParK* algorithm. All the vertices to be processed in a level or sub-level are distributed among the threads and the threads independently process the vertices. All the threads synchronize at the end of each sub-level. We have implemented our algorithm and experimented using different graphs, the largest of which has 65 million vertices and 1.8 billion edges. We compare our sequential algorithm with BZ algorithm. Our results show that the *ParK* algorithm is faster for all the graphs in our dataset (except one) and is more than 6 times faster for the largest graph in our dataset. We have also implemented the parallel version of the algorithm and the scalability and performance results are shown.

The rest of the paper is organized as follows. Section II provides some definitions and discusses the algorithm the BZ algorithm. Section III describes the *ParK* algorithm and the parallel methodology is explained in Section IV. The experimental study is presented in Section V and finally the concluding remarks are given in Section VI.

II. Background

Let $G = (V, E)$ be a graph where V is the set of vertices and E is the set of edges and let $n = |V|$ and $m = |E|$. The k-core of the graph G, is the largest induced subgraph in

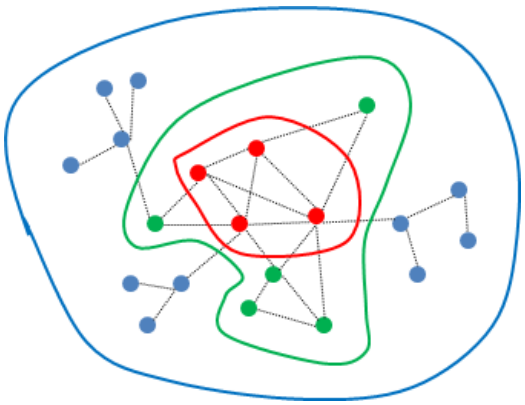


Figure 1. An example graph showing different cores

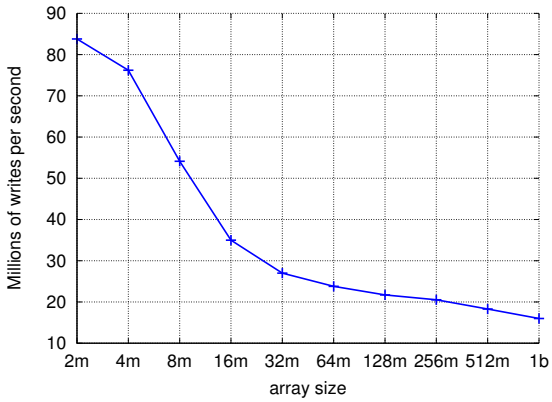


Figure 2. Plot showing the memory latency for random writes

which every vertex has degree at least k. The core number or *coreness* of a vertex, is the largest value of k for which the vertex belongs to the k-core. Note that $k+1$ core is a subset of k core. A k-shell of a graph G is the subgraph induced by the set of vertices in G whose core number is k, i.e. the vertices that belong to k-core but not $k+1$ core. In Figure 1, all the vertices inside the blue, green and red boundaries belong to 1-core, 2-core and 3-core respectively. The vertices colored in blue, green and red belong to 1-shell, 2-shell and 3-shell respectively and have core numbers 1, 2 and 3 respectively. The problem of k-core decomposition of the graph is to find all the k-cores of the graph or in other words, find the core numbers of all the vertices in the graph.

A. The algorithm of Batagelj and Zaversnik

To perform the k-core decomposition, the algorithm of Batagelj and Zaversnik (BZ) uses four arrays: deg, $vert$, pos and bin. The first three arrays are of size n and the bin array is of size M where M is the maximum degree of the graph. The deg array is initialized to contain the degrees of the vertices. i.e. $deg[v]$ contains the degree of vertex v where $0 \leq v < n$. The vertices are initially sorted by degree using bin-sort and sorted order is stored in the $vert$ array. The pos array stores the position of a vertex in the $vert$ array i.e. if $vert[i] = v$ then $pos[v]$ contains i. The bin array is initialized such that $bin[i]$ contains the index of the first vertex with degree i in the $vert$ array. For example, if the first 5 vertices in $vert$ array have degree 0 and the next 10 vertices have degree 1 then bin[0], bin[1] and bin[2] are initialized to 0, 5 and 15 respectively. The BZ algorithm is shown in Figure 3. The $vert$ array can be logically divided into chunks of vertices, chunk i consists of vertices with degree i where $0 \leq i \leq M$. The algorithm processes the vertices in $vert$ array in order. A vertex is processed by decrementing the degree of each of its neighbors and moving them to the appropriate chunks in the $vert$ array.

The theoretical complexity of BZ algorithm is shown to be $O(n+m)$ which makes it a very efficient algorithm for

```
1:  procedure kcore(deg, vert, pos, bin)
2:      for i = 0 to n − 1 do
3:          v = vert[i]
4:          for each vertex u adjacent to v do
5:              if deg[u] > deg[v] then
6:                  du = deg[u]
7:                  pu = pos[u]
8:                  pw = bin[du]
9:                  w = vert[pw]
10:                 if u ≠ w then
11:                     pos[u] = pw
12:                     pos[w] = pu
13:                     vert[pu] = w
14:                     vert[pw] = u
15:                 end if
16:                 bin[du] = bin[du] + 1
17:                 deg[u] = deg[u] − 1
18:             end if
19:         end for
20:     end for
21: end procedure
```

Figure 3. The algorithm of Batagelj et al.

k-core decomposition. However, observe that each iteration requires random read and write accesses to each of the deg, pos, bin and $vert$ arrays. Therefore, the working set includes all the four arrays. The random nature of the algorithm results incurs in large number of cache misses and hence high memory latency. To show the impact of random writes, we ran a benchmark which performs random writes to an integer array. The graph in Figure 2 clearly shows that the memory latency increases as the data size increases.

III. *ParK* ALGORITHM

We have seen that the BZ algorithm requires random access to multiple arrays and can be inefficient for large graphs. We propose a new algorithm which significantly

reduces the working set size and also the number of random accesses required. The outline of the algorithm is given in Figure 4. The algorithm uses three data structures: deg, $curr$ and $next$. deg is an array of size n initialized to contain the degrees of the vertices in the graph. It is updated as the algorithm advances and the values reflect the current core numbers of the vertices. $curr$ contains the set of vertices that are to be processed in the current iteration and next contains the vertices to be processed in the next iteration. The algorithm proceeds in levels. In level l all the vertices that belong to the l-shell are processed. Processing a vertex includes accessing all its neighbors and reducing their degree if they have not already been processed. Processing a level l is done in two phases: scan phase and loop phase. In scan phase, the deg array is scanned and the vertices that belong to the current level are collected in $curr$ i.e. $curr = \{v | deg[v] = l\}$. The loop phase consists of one or more sub-levels (or iterations). In each sub-level, all the vertices in $curr$ are processed. During processing of a vertex v if any of its neighbor u is moved to the current level i.e. $deg[u]$ is reduced to l, then u is added to $next$. At the end of the sub-level, contents of $next$ are transferred to $curr$ to be processed in the next sub-level. In Figure 4, the *ParK* algorithm is given in procedure *ParK* and the input deg array is initialized to the degrees of the vertices i.e. $deg[i]$ contains the degree of vertex i where $0 \leq i < n$. The scan phase is shown in procedure $scan$ and the procedure $processSublevel$ processes the vertices in $curr$ adding vertices to $next$ to be processed in the next sub-level.

```
1:  procedure scan(deg, level, curr)
2:      for i = 0 to n − 1 do
3:          if deg[i] = level then
4:              add i to curr
5:          end if
6:      end for
7:  end procedure
8:  procedure processSublevel(curr, deg, level, next)
9:      for each vertex v in curr do
10:         for each vertex u adjacent to v do
11:             if deg[u] > level then
12:                 decrement deg[u] by 1
13:                 if deg[u] = level then
14:                     add u to next
15:                 end if
16:             end if
17:         end for
18:     end for
19: end procedure
20: procedure ParK(deg)
21:     curr = ∅
22:     next = ∅
23:     todo = n
24:     level = 1
25:     while todo > 0 do
26:         scan(deg, level, curr)
27:         while |curr| > 0 do
28:             decrement todo by |curr|
29:             processSublevel(curr, deg, level, next)
30:             curr = next
31:             next = ∅
32:         end while
33:         increment level by 1
34:     end while
35: end procedure
```

Figure 4. *ParK* algorithm

A. Analysis of the algorithm

Lemma 1. The maximum number of levels is k_{max} where k_{max} is the largest value of k for which k-core is present in the graph.

Proof: The algorithm uses a variable $todo$ to keep track of the number of vertices to be processed. It can be seen from the procedure *ParK* in Figure 4 that $todo$ is decremented in each iteration by $curr$ which is the number of vertices processed in a sub-level. Therefore, $todo$ always contains the count of number of vertices to be processed. Since at the end of level k_{max} all the vertices have been processed and the value of $todo$ is zero, the maximum number of levels in k_{max}.

Lemma 2. The combined time taken for scan phase for all the levels is $O(k_{max}n)$

Proof: The scan phase in each level takes $O(n)$ and since there are k_{max} levels, the combined scan time for all the levels is $O(k_{max}n)$

Lemma 3. The combined time taken for sub-level processing (procedure *processSublevel* in Figure 4) is $O(m)$

Proof: In a single call to *processSublevel* a subset of vertices is processed. Combinedly in all the calls to *processSublevel*, all the n vertices are processed. Note that, each vertex is processed exactly once i.e. when its degree becomes equal to the current level. We have seen that processing a vertex v includes reducing the degree for each of its neighbor if it has not already been processed which takes $O(d_v)$ time where d_v is the degree of vertex v. Therefore, to process all the n vertices it takes $O(m)$ time.

Combining lemmas 2 and 3, the computational complexity of *ParK* algorithm is $O(k_{max}n + m)$. Though the computational complexity of BZ algorithm, which is $O(m)$, is less compared to the *ParK* algorithm, our experimental results show that *ParK* algorithm outperforms BZ algorithm. The reason is that by using the scan phase, *ParK* significantly reduces the working set size. The BZ algorithm requires random read and write accesses to three different arrays of size n while in *ParK* algorithm requires random access to

```
1: procedure scan(deg, level, curr)
2:     idx = 0
3:     for i = 0 to n − 1 do in parallel
4:         if deg[i] = level then
5:             a = atomicIncrement(idx, 1)
6:             curr[a] = i
7:         end if
8:     end for
9: end procedure
10: procedure processSublevel(curr, deg, level, next)
11:     idx = 0
12:     for each vertex v in curr do in parallel
13:         for each vertex u adjacent to v do
14:             if deg[u] > level then
15:                 a = atomicDecrement(deg[u], 1)
16:                 if a ≤ level then
17:                     atomicIncrement(deg[u], 1)
18:                 end if
19:                 if a + 1 = level then
20:                     b = atomicIncrement(idx, 1)
21:                     next[b] = u
22:                 end if
23:             end if
24:         end for
25:     end for
26: end procedure
```

Figure 5. Parallel version of *ParK* algorithm

only one array. Though the theoretical time taken for the scan phase seems significant, in practice the time taken for scan phase is less compared to the time saved due to the scan phase. Another major advantage of scan phase is that it is embarrassingly parallel. It can easily and efficiently be distributed among different processors and can scale linearly.

IV. Parallel Methodology

The *ParK* algorithm given in Figure 4 is described in sequential context. In this section we provide the details of parallel approach. Parallelizing the *ParK* algorithm is done by parallelizing the two major components of the *ParK* algorithm: *scan* and *processSublevel*. Parallelizing the scan phase is simple and trivial. The n vertices are equally divided among the t threads and each thread scans n/t vertices. Note that distributing the n vertices among t threads can be done in several ways. To minimize cache misses contiguous chunks of vertices are assigned to the threads. To process each vertex, a thread reads its degree and if it is equal to the current level it adds the vertex to $curr$ array. Since number of operations performed for each vertex is very less, all the threads take almost same time resulting in linear speedup.

Though, it is simple to parallelize the scan phase, there is one issue to be addressed. It is to be noted that, $curr$ is shared between all the threads and multiple threads writing to it may result in race conditions. The modified version of $scan$ procedure is shown in Figure 5. The function $atomicIncrement(idx, 1)$ increments the value of idx by 1 and returns its old value atomically (implemented using *atomic capture* construct in OpenMP). Using atomic operations the race conditions are eliminated. However, the atomic operations are expensive and too many atomic operations can significantly downgrade the performance. To reduce the number of atomic operations, the vertices are added in batches instead of a single vertex. We use a local buffer of size b. Instead of adding each vertex to $curr$, they are first added to the local buffer. When the buffer is full, idx is atomically incremented by b and all the vertices are transferred from local buffer to $curr$. This reduces the number of atomic operations by a factor of b. Note that, to avoid cache invalidation the size of local buffer should be a multiple of cache line size.

The next component to be parallelized is *processSublevel*. We have seen that in *processSublevel* all the vertices in $curr$ are processed i.e. the degree of all their neighbors are reduced and if any of the neighbors belong to the current level, it is added to $next$. Parallelizing the procedure is done by equally distributing the vertices in $curr$ to all the threads. However, it might result in race conditions as all the threads are reading from and writing to deg array and $next$. To avoid race conditions, the deg array is updated atomically. However, it is possible that the degree of a neighbor u, i.e. $deg[u]$ is reduced to a value less than the current level value. For example, let $deg[u] = level+1$ and two or more threads execute the line 14 at the same time, test positive for the condition and execute line 15. The resultant value of $deg[u]$ will be less than $level$ which is wrong as u belongs to the current level and $deg[u]$ should be equal to $level$. This is fixed using lines 16 through 18. Vertices are added to $next$ similar to the way vertices are added to $curr$ in the *scan* phase i.e. using atomic increments and local buffers.

The *ParK* procedure shown in Figure 4 can be parallelized by simply adding synchronization calls after *scan* and *processSublevel* phases. We use OpenMP *barrier* construct to synchronize the threads. Note that the instructions from line 23 to 34 are part of parallel region and the lines 30 and 31 must be executed by a single thread.

V. Experimental Results

All the results we present in this paper are obtained on a four socket 2.27GHz Xeon X7560(Nehalem-EX) with 256 GB shared memory and running 64-bit Ubuntu 12.04. Each socket consists of 8 cores. Each core has a private 32 KB L1 cache and 256 KB L2 cache. A 24 MB L3 cache is shared by all the cores in a socket. All the implementation is done using C programming language and compiled using gcc compiler with -O3 optimization flag. The parallel implementation is done using OpenMP. Our dataset consists of several graphs from the Stanford Large Network

Collection [26] and random graphs. We use four random graphs generated using GTGraph, a graph generator[27]. *rand-er-1m* has 1 million vertices and is generated using ErdosRenyi graph model with probability 10^{-3}. *rmat-32-256* and *rmat-32-512* graphs are generated using the R-MAT model. We used default parameter values in GTgraph i.e. $(a, b, c, d) = (0.45, 0.15, 0.15, 0.25)$. *rand-32-512* is a random graph generated by adding each edge to a randomly chosen pair of vertices. The details of the graphs are shown in Table I. The number of vertices and edges is given in millions.

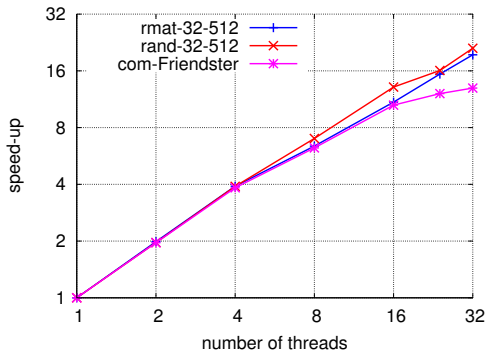
In Table I, we compare the timing results of sequential *ParK* algorithm (in Figure 4) with the state-of-the-art BZ algorithm. All the results shown refer to time in seconds. Note that, the results shown include the time taken for initialization. For the *ParK* algorithm, we show total running time and the time taken for scan phase. It can be seen that the *ParK* algorithm is faster than the BZ algorithm for all the graphs, except *wiki-Talk* graph. Also notice that the *ParK* algorithm outperforms BZ algorithm by a larger factor as the graphs size increases. The main reason contributing to this is the relatively low working set size and hence improved locality of reference. To verify this we checked the cache performance using Cachegrind[28]. For *soc-LiveJournal1* graph, the BZ algorithm resulted in 227 million(148m read + 79m write) L1 data cache misses and 46m(31m read + 15m write) L3 cache misses while the sequential *ParK* resulted in 202m(192m read + 10m write) L1 cache misses and 32m(22m read + 10m write) L3 cache misses. Notice the time taken for the scan phase. It is significantly low for smaller graphs and increases with the graphs size. However, since the scan phase is embarrassingly parallel, it is not an issue when multiple threads are used. *ParK* performs slower than BZ algorithm in the case of *wiki-Talk* graph. We observed that in the *wiki-Talk* graph, there are large number of shells with only few vertices in each shell resulting in large number of levels. In this case, the time spent in scan phase is significant. However, this problem can be mitigated using multiple threads since the scan phase scales linearly.

We measure the performance in terms of millions of edges per second which is computed using $m_a/time$ where m_a is the number of edges accessed and $time$ is the running time of the algorithm in seconds. Since the *ParK* algorithm processes all the vertices exactly once by accessing all of their neighbors, each edge in the graph is accessed exactly twice and so $m_a = 2m$. For the results to correctly reflect the performance, we exclude the initialization time from the running time. To avoid unexpected behavior, for all our experiments we pin the threads to specific cores such that threads 0 to 7, 8 to 15, 16 to 23, 24 to 31 run on 1st, 2nd, 3rd and 4th socket respectively.

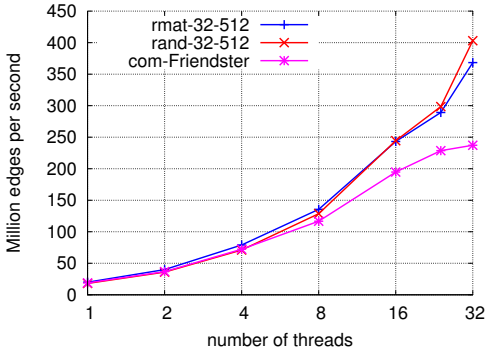
Figures 6a and 6b plot the speedup and processing rates respectively for three graphs: *rmat-32-512*, *rand-32-512* and *com-Friendster*. We can see that the approach scales well

Table I
DETAILS OF GRAPHS AND COMPARISON BETWEEN BATAGELJ ET AL. ALGORITHM AND *ParK* ALGORITHM

graph	n	m	k_{max}	BZ	*ParK* $total$	*ParK* $scan$
amazon0601	0.4	2.4	10	0.16	0.09	0.005
web-BerkStan	0.6	6.6	201	0.23	0.22	0.11
web-Google	0.9	4.3	44	0.31	0.19	0.03
wiki-Talk	2.4	4.6	131	0.35	0.45	0.25
as-Skitter	1.7	11.1	111	0.84	0.49	0.16
soc-Pokec	1.6	30.6	47	2.29	0.83	0.07
cit-Patents	3.8	16.5	64	3.42	1.44	0.22
rand-er-1m	1.0	95.0	160	4.36	1.85	0.12
com-Orkut	3.1	117.2	253	18.02	5.48	0.64
soc-LiveJournal1	4.8	69.0	362	5.47	3.04	1.42
rmat-32-256	32.0	256.0	29	147.67	24.16	1.86
rmat-32-512	32.0	512.0	59	288.1	41.17	3.65
rand-32-512	32.0	512.0	23	231.38	51.5	1.47
com-Friendster	65.6	1806.0	289	981.58	158.68	36.07



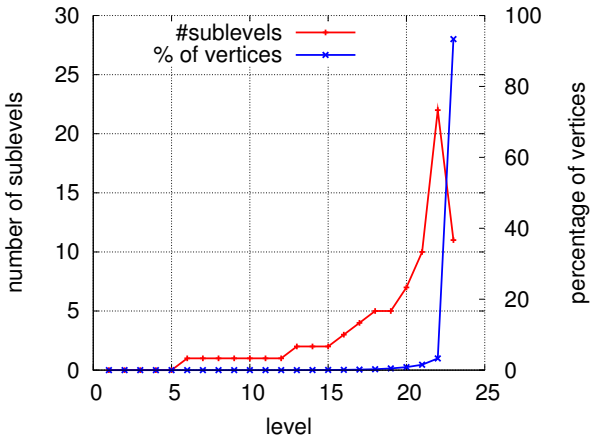
(a) speed-up



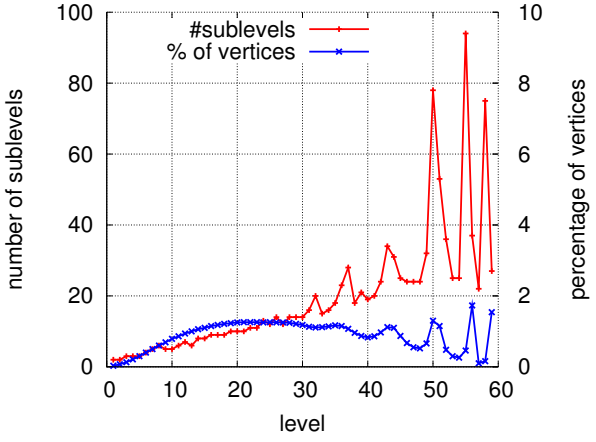
(b) processing rates

Figure 6. Scalability and Performance results for different graphs

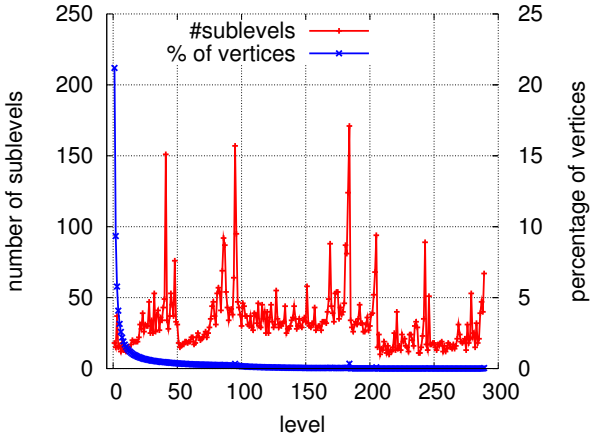
for both *rmat-32-512* and *rand-32-512* graphs. Also, the processing rate increases gradually with the increase in number of threads. However, for the *com-Friendster* graph, the approach does not scale considerably and shows only limited growth in processing rate. We closely analyze the graphs to understand the behavior of the approach. We consider the following factors to analyze a graph: number of levels, number of sub-levels and percentage of vertices processed in each level. We have seen that the task of processing the vertices in sub-level is distributed among



(a) rand-32-512



(b) rmat-32-512



(c) com-Friendster

Figure 7. Plots showing the number of sub-levels in a level and the percentage of vertices processed in a level

multiple threads and the threads are synchronized after each sub-level. If number of vertices processed in a level is large, the threads spend most of the time in processing the vertices and the synchronization overhead incurred after every sub-level is minimum. However, if there are only a few vertices in a level and the number of sub-levels is large, the threads spend most of their time at the synchronization barrier resulting in huge synchronization overhead.

Figure 7 plots the number of sub-levels and percentage of vertices processed in each level. Note that the missing percentage of vertices in the plots belong to level 0. In *rand-32-512* graph, most of the levels have low percentage of vertices. However, since these levels have only few sub-levels, the synchronization overhead is minimum and so the approach is able to scale well. In case of *rmat-32-512* graph, all the levels process approximately the same number of vertices (around 1 to 2 percent which is 320,000 to 640,000) which is large enough to keep the threads busy for longer time than the synchronization time. And, there are only few levels with large number of sub-levels. Therefore, the approach achieves good speed-up for the graph. In *com-Friendster* graph, however, the majority of the vertices are processed in the lower levels (level less than 50). Interestingly, the number of sub-levels is also low for these levels. All the remaining levels (beyond 50), process very few vertices and large number of sub-levels. This incurs in huge synchronization overhead justifying the speedup and processing rate shown in Figure 6.

The parallelization methodology used in this paper can be improved further by reducing the synchronization overhead caused due to large number of sub-levels. Since *ParK* algorithm is a level-synchronous approach similar to the BFS algorithm of Agarwal et al., the parallelization techniques used in [20] can be applied to *ParK*. For example, as the inter-socket atomic operations cannot scale efficiently across sockets, they are avoided using a channel mechanism. The vertices can be divided among the sockets and the threads process only the vertices that are assigned to the socket in which the thread is running. Any vertex that is to be processed and is assigned to other socket is placed in a socket queue of the corresponding socket. This confines the atomic operations to the sockets.

VI. Conclusion

In this paper, we presented a new algorithm for k-core decomposition that is efficient and is also amenable to parallelization. The algorithm significantly reduces the working set size and the number of random accesses compared to the state-of-the-art algorithm for k-core decomposition. We present a simple parallelization methodology focusing on multicore processors. We have implemented the algorithm and tested the performance using various graphs with up to 65 million vertices and 1.8 billion edges. We show that the algorithm outperforms the state-of-the-art algorithm, more

than 6 times for the largest graph used in the dataset. We also present the scalability and performance results for some of the graphs and show that the approach scales well for some of the graphs. The parallel methodology given in this paper can be improved using the techniques used for a level-synchronous approach.

REFERENCES

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

[2] S. Wasserman and K. Faust, *Social network analysis: Methods and applications*. Cambridge university press, 1994, vol. 8.

[3] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases," *NHM*, vol. 3, no. 2, pp. 371–393, 2008.

[4] S. Carmi, S. Havlin, S. Kirkpatrick, and E. Shir, "Medusa - new model of internet topology using k-shell decomposition," *PNAS*, pp. 11–150, 2007.

[5] L. Gallos, S. Havlin, M. Kitsak, F. Liljeros, H. Makse, L. Muchnik, and H. Stanley, "Identification of influential spreaders in complex networks," *Nature Physics*, vol. 6, no. 11, pp. 888–893, Aug 2010.

[6] M. Pellegrini, F. Geraci, and M. Baglioni, "Detecting dense communities in large social and information networks with the core & peel algorithm," *CoRR*, vol. abs/1210.3266, 2012.

[7] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos, "Community detection in social media," *Data Mining and Knowledge Discovery*, vol. 24, no. 3, 2012.

[8] V. Batagelj, A. Mrvar, and M. Zaversnik, "Partitioning approach to visualization of large graphs," in *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*. London, UK: Springer-Verlag, pp. 90–97.

[9] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: a tool for the visualization of large scale networks," *CoRR*, vol. abs/cs/0504107, 2005.

[10] J. I. Alvarez-hamelin, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Advances in Neural Information Processing Systems 18*. MIT Press, 2006, pp. 41–50.

[11] G. D. Bader and C. W. Hogue, "Analyzing yeast protein-protein interaction data obtained from different sources." *Nature biotechnology*, vol. 20, no. 10, pp. 991–997, 2002.

[12] Y. Cheng, C. Lu, and N. Wang, "Local k-core clustering for gene networks," in *IEEE International Conference on Bioinformatics and Biomedicine*, 2013, pp. 9–15.

[13] D. Eppstein, M. Lffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time," in *ISAAC*, ser. Lecture Notes in Computer Science, 2010, vol. 6506, pp. 403–414.

[14] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, and M. M. A. Patwary, "A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components," *CoRR*, vol. abs/1302.6256, 2013.

[15] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006.

[16] C. Staudt, A. Sazonovs, and H. Meyerhenke, "Networkit: An interactive tool suite for high-performance network analysis," *CoRR*, vol. abs/1403.3005, 2014.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010. [Online]. Available: http://arxiv.org/abs/1006.4990

[18] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38 – 53, 2009.

[19] M. Horton, S. Tomov, and J. Dongarra, "A class of hybrid lapack algorithms for multicore and gpu architectures," in *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, ser. SAAHPC '11, 2011, pp. 150–158.

[20] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, 2010, pp. 1–11.

[21] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 78–88.

[22] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proc. VLDB Endow.*, vol. 6, no. 6, pp. 433–444, Apr. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536336.2536344

[23] R. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2453–2465, 2014.

[24] D. Miorandi and F. D. Pellegrini, "K-shell decomposition for dynamic complex networks," in *WiOpt*, 2010, pp. 488–496.

[25] A. Montresor, F. D. Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 2, pp. 288–300, 2013.

[26] J. Leskovec, "Stanford Large Network Dataset Collection." [Online]. Available: http://snap.stanford.edu/data/

[27] K. Madduri and D. A. Bader, "GTgraph: A suite of synthetic graph generators," accessed: 12.02.2014. [Online]. Available: https://github.com/dhruvbird/GTgraph

[28] J. Seward, N. Nethercote, and J. Weidendorfer, *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008.