2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Parallel k-core Decomposition on Multicore Platforms

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

***Abstract*—k-core decomposition is a social network analytic that can be applied to centrality analysis, visualization, and community detection. There is a simple and well-known linear time sequential algorithm to perform k-core decomposition, and for this reason, k-core decomposition is widely used as a network analytic. In this work, we present a new shared-memory parallel algorithm called PKC for k-core decomposition on multicore platforms. This approach improves on the state-of-the-art implementations for k-core decomposition algorithms by reducing synchronization overhead and creating a smaller graph to process high-degree vertices. We show that PKC consistently outperforms implementations of other methods on a 32-core multicore server and on a collection of large sparse graphs. We achieve a 2.81× speedup (geometric mean of speedups for 17 large graphs) over our implementation of the ParK k-core decomposition algorithm.**

***Keywords*-k-core; multicore; network analysis**

I. Introduction

Networks are used to model interacting entities, and network models and algorithms help analyze social data, large-scale data in science and engineering, as well as data from sensors and the internet. There are several useful graph-theoretic notions that have been adopted by the network analysis community. One such computation is k-core decomposition [1], [2].

The k-core of a graph is a maximal subgraph in which all vertices have degree at least k. The problem of k-core decomposition refers to finding the *coreness* of every vertex in the graph. A vertex has coreness l if it belongs to an l-core but not to an $(l+1)$-core. Linear time algorithms are known for performing k-core decomposition. This makes k-core decomposition very useful as a generic network analysis tool.

One of the early applications of k-core decomposition was in visualization of large networks [3]. In this work, vertices are assigned coordinates on concentric cycles based on their k-core values. Vertices with higher coreness values are placed closer to the center. Additionally, if vertices from different connected components are in same k-core, they are on same concentric cycle, but put in different clusters.

k-core decomposition has also been applied to analyze networks in bioinformatics [4]. Clusters are found in a human gene network data using k-core decomposition. The k-cores may not however represent good clusters. So, the authors extend the k-core concept to use an additional parameter r. A method proposed in [4] finds a subgraph that is a k-core and also an r-clique. Thus, a k-core also participates in complete subgraphs of size r.

k-core decomposition has been used to analyze large-scale internet maps [5]. It is observed that the k-core subgraphs are connected in these networks. The k-core subgraphs exhibit a hierarchical structure, i.e., the k-core is a fraction of the $(k-1)$-core, and the size of k-core subgraphs in terms of vertices are shown to follow a power law. k-core decomposition is also used as a preprocessing strategy for solving NP-hard problems, for example, finding the maximum clique in a graph [6].

Peng et al. use k-core decomposition to accelerate community detection algorithms [7]. Community detection algorithms take at least time linear in graph size (number of vertices and edges). These algorithms are accelerated by applying them on k-core subgraphs instead of the full graph. An assumption in this work is that communities are preserved in k-core subgraphs [7].

The k-core concept has been extended to directed graphs (called a D-core) to find communities [8]. The communities found by a D-core are closely connected. A generalized k-core decomposition approach is proposed in [9], which considers both degree and weight for a vertex. It is observed that unweighted k-core decomposition is a special case of weighted k-core decomposition. The generalized k-core decomposition algorithm is applied to different weighted networks, and this partitions a network in a more refined and meaningful way. The authors show that weighted k-core puts vertices with higher spreading potential near the core (maximum k-core) of a network [9].

k-core analysis clearly has many uses. In this paper, we present a new shared-memory parallel algorithm for k-core decomposition. In our algorithm, coreness values of vertices are computed in a level-synchronous manner. We evaluate this algorithm and other state-of-the-art algorithms on a 32-core Intel multicore server.

II. Background and Related Work

Let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and $m = |E|$ edges. A k-core of the graph G is defined as a maximal induced subgraph of G such that each

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.151

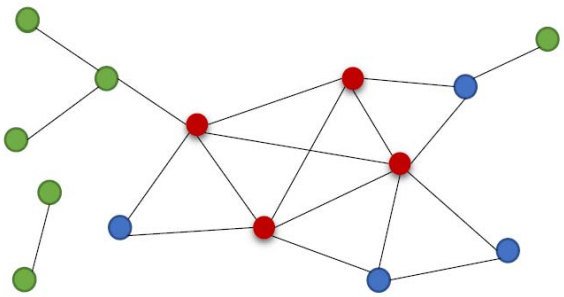

Figure 1. An example graph to illustrate k-core decomposition. Vertex colors indicate coreness values (1: green, 2: blue, 3: red).

vertex in this subgraph has degree at least k. The k-core value or the coreness value of a vertex is defined as the k-core with the largest value of k that it belongs to. Let K_{max} denote the maximum coreness value of any vertex in the graph. In Figure 1, the coreness values of the vertices in a small graph are shown. The green vertices have a coreness value of 1, the blue vertices have a coreness value of 2, and the red vertices have a coreness value of 3. In this example, $K_{\text{max}} = 3$. Note that a k-core subgraph can have multiple connected components. In this work, when we say we are performing the k-core decomposition, we refer to computing the coreness values of all the vertices. The problem of identifying connected components within k-cores is related, but requires an additional graph traversal.

A. Related Work

To compute the k-core decomposition of a graph, a popular algorithm is one proposed by Batagelj and Zaversnik [10]. We refer to this algorithm as BZ in subsequent discussion. This algorithm is used in practice because its asymptotic running time bounds are $O(n + m)$. The BZ algorithm uses the property that if all the vertices of degree less than k and the edges incident on those vertices are deleted from the graph, then the remaining graph is a k-core [10], and the coreness values of vertices can be set appropriately. The BZ algorithm uses two arrays of size n, one to sort the vertices in increasing degree (using a linear time sort), and the other to record the positions in sorted order. Vertices are visited according to increasing degree, and if any adjacency has degree more than the currently-visited vertex, then the degree of the neighbor is decreased and the adjacency's bin is updated. This is done by swapping the neighbor with the first vertex in the bin containing the neighbor. Next, the neighbor is put in the previous bin and the arrays are updated to maintain the order of the vertices in increasing degree.

Montresor, De Pellegrini, and Miorandi propose a distributed algorithm for k-core decomposition [11] that is very different from BZ. We call this the MPM algorithm. MPM can be used when the graph is partitioned and stored on multiple compute systems. In this algorithm, each processor is responsible for computing coreness values of a subset of vertices. Each processor has a subset of vertices and the adjacencies of these vertices. Initially, a processor assigns the degree of a vertex as its core value, and sets the core values of non-local adjacencies to infinity. Processors perform a local update step where coreness estimates are updated, and each processor propagates the updated core values for the subset of vertices it owns to its neighbor processors. The neighbor processors use the received core values and again update coreness estimates. The update process continues until convergence. The authors give simple pseudocode for the local update step and demonstrate that the algorithm converges in practice in fewer iterations than the maximum coreness value. However, in the worst case, the algorithm requires $O(n)$ iterations for convergence, and each iteration could take $O(m)$ time, and so the time complexity is $O(nm)$. In practice, the algorithm converges in $it_{\text{MPM}} < K_{\text{max}} \ll n$ rounds, so time complexity is $O(it_{\text{MPM}}m)$. We develop a shared-memory implementation of MPM and compare the performance of our algorithm with MPM.

Khaouid et al. [12] study algorithms and implementations for k-core decomposition on low-memory systems. The implementations in [12] use the GraphChi [13] graph processing system and the WebGraph compression library [14]. An implementation of the BZ algorithm using WebGraph is also presented. The authors show that the BZ algorithm outperforms other algorithms in a sequential setting and when the graph fits in main memory. If the graph is too big to store in memory, an external memory algorithm EM-core [15] is used to compute k-core values. The authors compare a baseline EM-core approach to an optimized approach using GraphChi and WebGraph.

Dasari, Ranjan, and Zubair propose ParK [16], a shared-memory k-core decomposition algorithm for multicore platforms. They show that the BZ algorithm incurs a high number of random memory references, and this affects the performance of BZ on multi-socket systems. We discuss the ParK algorithm in the next section, as our proposed approach is closely related to this algorithm.

k-core decomposition is used as a subroutine in a parallel algorithm to find the maximum clique in a graph [6]. The coreness values are used to find the upper bound on the size of a maximum clique. In the pmc tool [6], the sequential BZ algorithm is used to compute the k-core decomposition of a graph, and this takes a significant amount of time in the overall execution. We show that replacing the BZ implementation by our approach can improve overall performance of the pmc tool on several large-scale graphs.

Modern computing systems have large main memory capacities. The memory capacity enables us to process extremely large graphs on a single system. For instance, in

this study we use a system with 512 GB main memory. On a shared memory system, we can achieve good performance even without explicit partitioning of the graph or out-of-core storage of the graph structure.

III. Parallel k-core Decomposition

In this section, we present our algorithm PKC. We start by describing the ParK algorithm because it is similar to our proposed algorithm.

A. The ParK algorithm

In the ParK [16] algorithm, vertices are processed level by level: i.e., all the vertices belonging to an l-core are first identified before finding vertices that belong to an $(l+1)$-core. The steps of the ParK algorithm are listed in Algorithm 1. The algorithm uses three shared arrays deg, $curr$, and $next$ of size n (where n is the number of vertices in the graph). The deg array is initialized to the degree of the vertices, i.e., $deg[i]$ is the degree of a vertex i and reflects the current coreness estimate of a vertex. Note that the vertex degree is an upper bound on the coreness value.

To find the vertices belonging to a level l (i.e., l-core), the algorithm uses two subroutines, Scan and ProcessSubLevel. In Scan, vertices are visited in order, and all vertices whose current core value is equal to l are added to the $curr$ array. Now the vertices in the $curr$ array are processed in the ProcessSubLevel phase. The vertices of $curr$ are processed by inspecting their adjacencies. Processing an adjacency may involve decreasing the core value of this adjacency. Also, an adjacency may be added to the $next$ array if its core value becomes equal to l. The $next$ array is processed in the next iteration. This is done by swapping $next$ and $curr$ array and processing the new $curr$ array.

The sequential running time bounds of ParK are inferior to the linear-time BZ. If $K_{\max}$ is the largest coreness value any vertex in the graph can take, the Scan procedure is executed $K_{\max}$ times in ParK and each scan takes $O(n)$ time. A vertex is processed exactly once, and so the total time needed to process the adjacencies of all vertices is $O(m)$ (where m is the number of edges in the graph). Thus, the time complexity of ParK is $O(nK_{\max}+m)$.

The parallelized steps in ParK are indicated in the pseudocode. The Scan procedure is easy to parallelize. The current core values of vertices are inspected, and if the core value is equal to current level, they are added to $curr$. Since all the threads are adding to the same array $curr$, the vertices should be added atomically to the array.

Parallelizing procedure ProcessSubLevel is a bit tricky. The vertices in the $curr$ array are processed and the vertices to be processed in the next iteration are added to the $next$ array. The $curr$ array is processed in parallel and the neighbors are visited. A neighbor may be visited by multiple threads, and the degree of a neighbor may be decreased by multiple threads. That is why the degree must be decreased atomically. Also, to decrease the degree of a vertex, more than one thread may evaluate the condition that the degree of a vertex is greater than current level. If more than one thread

Algorithm 1 Our implementation of the ParK algorithm.

```
procedure SCAN(deg, l, curr)
    Initialize a thread-local array buff of size s
    i ← 0                                   ▷ thread-local variable
    for (v = 0 to n − 1) in parallel do
        if (deg[v] = l) then
            buff[i] ← v; i ← i + 1
            if (i = s) then
                Atomically update end of curr
                Copy buff to curr
                buff ← φ; i ← 0
    if (i > 0) then
        Atomically update end of curr
        Copy buff to curr
        buff ← φ; i ← 0

procedure PROCESSSUBLEVEL(curr, deg, l, next)
    Initialize a thread-local array buff of size s
    i ← 0
    for (v ∈ curr) in parallel do
        for (u ∈ Adj(v)) do
            if (deg[u] > l) then
                a ← atomicSub(deg[u], 1)
                if (a = (l + 1)) then
                    buff[i] ← u; i ← i + 1
                    if (i = s) then
                        Atomically update end of next
                        Copy buff to next
                        buff ← φ; i ← 0
                if a ≤ l then
                    atomicAdd(deg[u], 1)
    if (i > 0) then
        Atomically update end of next
        Copy buff to next
        buff ← φ; i ← 0

procedure PARK(deg)
    curr ← φ; next ← φ
    todo ← n; l ← 0
    while todo > 0 do
        SCAN (deg, l, curr)
        while |curr| > 0 do
            todo ← todo − |curr|
            PROCESSSUBLEVEL(curr, deg, l, next)
            curr ← next
            next ← φ
        l ← l + 1
```

decreases the degree, the degree may decrease below the current level. This is corrected by incrementing the degree if it goes below the current level. The vertices are added atomically to the $next$ array by the threads. This procedure ensures that only one thread adds a vertex to the $next$ array.

The compute-intensive steps are all executed in parallel. Swapping of $curr$ and $next$ is done by a single thread. ParK also uses synchronization calls (barrier) after SCAN, PROCESSSUBLEVEL, and after swapping the arrays.

To optimize performance of ParK, the vertices in the SCAN and PROCESSSUBLEVEL phases are not added directly to the $curr$ and $next$ arrays, respectively. Instead, each thread uses a fixed size buffer of size s and adds the vertices to the buffer first. The vertices are copied to the corresponding array once the buffers are full. This decreases the number of atomic operations needed to add vertices to the global arrays.

The ParK algorithm performs quite well in practice, as $K_{\max}$ is much smaller than the maximum vertex degree. However, we can further decrease the number of atomic operations, and the number of synchronization calls (barriers). We observe that in Algorithm 1, the number of synchronization calls is equal to $K_{\max} + 2S$, where $S = \sum_{i=1}^{K_{\max}} nsl[i]$ and $nsl[i]$ is the number of sub-levels at level i. We also observe that a vertex is added only once to the $curr$ array. This addition is done atomically. The number of atomic operations needed to add n vertices is n/s, where s is the size of the buffers used by the threads. In addition, we observe that even though the number of levels $K_{\max}$ in a graph could be very high, most of the vertices belong to lower cores. We propose an algorithm that decreases the number of synchronization calls, the number of atomic operations, and exploits the property that most vertices of the graph have low coreness values.

B. PKC

We describe our approach PKC in this subsection. We observed that in Algorithm 1, two arrays of size n ($curr$ and $next$) are used, in addition to deg to store the core values of the vertices. There are synchronization calls after each call to SCAN and PROCESSSUBLEVEL, and after swapping $curr$ and $next$. We will decrease these overheads in our approach. We first describe a baseline version of PKC and then discuss an additional optimization to reduce extraneous work performed.

To compute k-core decomposition of a graph, we observe that we do not need to use $curr$ and $next$ array and swap them after every call to PROCESSSUBLEVEL. Consider a sequential version of PKC approach listed in Algorithm 2. If there is just one thread, only one array $buff$ of size n is used, in addition to deg to hold the core values. To process the vertices in level l, we scan the deg array, and if the degree of a vertex is equal to l, it is added to the array

Algorithm 2 Our PKC algorithm.

```
procedure PKC(deg)
    i ← 0                                              ▷ Global var
    l ← 0; s ← 0; e ← 0                                ▷ Thread-local var
    Initialize a thread-local array buff of size n/n_t
    while (i < n) do
        for (v = 0 to n − 1) in parallel do
            if (deg[v] = l) then
                buff[e] ← v; e ← e + 1
        while (s < e) do                               ▷ process local buff
            v ← buff[s]; s ← s + 1
            for (u ∈ Adj(v)) do
                if (deg[u] > l) then
                    a ← atomicSub(deg[u], 1)
                    if (a = (l + 1)) then
                        buff[e] ← u; e ← e + 1
                    if (a ≤ l) then
                        atomicAdd(deg[u], 1)
        Barrier synchronization
        atomicAdd(i, e)
        s ← 0; e ← 0; l ← l + 1
```

$buff$. Next, the vertices in $buff$ are processed in a loop, until the buff becomes empty. A vertex v of $buff$ is processed by processing its neighbors. The current core value of the neighbors of v are checked and if the current core value is bigger than l, the core value of a neighbor is decreased. This may put the neighbor in the current core l. This is done by adding the neighbor to the same array $buff$. The processing of vertices of $buff$ stops when no vertices are left to process. Once the processing of level l completes, the algorithm proceeds to process vertices belonging to level $l + 1$ and it terminates when all the vertices of a graph are processed.

The sequential time complexity of PKC is same as ParK algorithm. In PKC, the deg array is scanned $K_{\max}$ times and each scan takes $O(n)$ times. Thus the scan time is $O(nK_{\max})$. Each vertex is processed exactly once by visiting the adjacencies of the vertices, which takes $O(m)$ time. So the time complexity of PKC is $O(nK_{\max} + m)$.

Now consider the parallel approach. We assign an array $buff$ of size n/n_t to each thread, where n_t is the number of threads used. In the first part, the array deg is scanned in parallel by the threads and the thread processing a vertex adds it to its buffer if the current core is equal to current level. This does not need any atomic operations, as the threads add vertices to their local buffers.

The next phase in the algorithm is to process the vertices in the $buff$ of each thread. Each thread processes the vertices in its local $buff$. However, two threads processing vertices in their local buffer may need to process a neighbor at the same time. Thus, if the current core value is greater

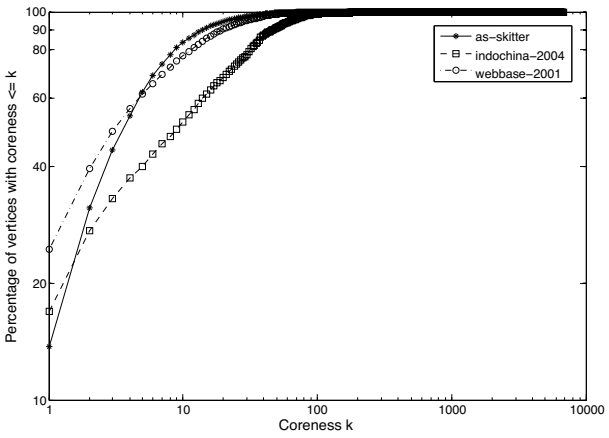

Figure 2. The coreness distribution of vertices in each level for three graphs. More than 50% of vertices have coreness values less than 10 on all three graphs. Note the log-log axes.

than the level, it is decremented atomically. Also, the core value of a neighbor may be decreased below current level if more than one thread is processing the neighbor. This is corrected by increasing the core value if any core value goes below current level. The threads add the neighbors to its local buffer if any of the neighbor becomes part of current level. The processing finishes when the $buff$ of every thread becomes empty.

The pseudocode indicates the loops that are executed in parallel. The algorithm also needs a synchronization call after each level. The algorithm terminates when all the vertices in a graph are processed.

Each thread in Algorithm 2 uses an array of size n/n_t. Thus our algorithm uses just one additional array of size n, in addition to deg to hold core values of the vertices. The memory usage is lower than Algorithm 1, because Algorithm 1 uses two arrays of size n and a constant size buffer on each thread, in addition to deg array. We observe that our algorithm uses only one synchronization call in each level, and in total, it uses $K_{\max}$ barriers. This could be far lower than $K_{\max} + 2S$ (S is the sum of sub-levels), if there are many sub-levels in each level of the graph. Additionally, our algorithm decreases the number of atomic operations. In Algorithm 1, n/s atomic operations are used to add the vertices to $curr$ array. Algorithm 2 does not need these atomic operations, as each thread is adding the vertices to its local $buff$.

Optimizations in PKC. We can further improve Algorithm 2, by decreasing the time taken to scan the deg array and processing the sublevels. The SCAN time $O(K_{\max}n)$ could be substantial if $K_{\max}$ is large. Also, for most of the graphs, k-core decomposition time is dominated by the time taken to process the vertices by threads contained in their individual $buff$. Processing time is more noticeable for high-degree vertices in graphs with skewed degree distributions. This is the case for the vertices belonging to the higher cores, because high degree vertices tend to have high coreness values.

Figure 2 shows the percentage of vertices processed in each level for three graphs. It can be observed that most of the vertices have low coreness values. For example, $K_{\max}$ for indochina-2004 is 6869, but more than 98% of the vertices belong to first 87 levels or cores. The remaining 2% vertices belong to rest of the levels and they have higher degree. These vertices are processed by checking all their adjacent vertices, even though some of the adjacencies belong to lower cores.

We exploit this observation to improve PKC. Once a significant fraction f of the vertices are processed, we create a new array $newDeg$ containing the current core estimates of the remaining vertices. For the remaining levels, this new array of significantly smaller size is scanned. If this switch is done at level l, the SCAN time to process the remaining vertices is $O((K_{\max} - l)(1 - f)n)$. This is lower than $O((K_{\max} - l)n)$ by a factor of $\frac{1}{1-f}$.

To decrease the processing time, we create a new graph ($newG$) with the the remaining vertices, $(1 - f)n$. There is an edge $\langle u, v\rangle \in newG$ if $\langle u, v\rangle$ is an edge in the original graph and both u and v have coreness values larger than l. After scanning $newDeg$ array, the threads process the vertices using the graph $newG$. The number of adjacencies of a vertex in $newG$ could be smaller than the original graph, because $newG$ only contains the remaining vertices. The graph $newG$ is created in parallel and once the cores are computed using this graph, the core values are copied from $newDeg$ to deg in parallel.

The optimized algorithm improves over Algorithm 2. The overall improvement depends on the values of l and f. We want l to be small and f to be high (as close to 1 as possible). If we pick one, the other is automatically set. We choose to fix f to 0.98 in this work. The process phase is also improved, because $newG$ is a small graph and number of adjacencies are lower in this graph. Also, this may increase locality in memory access pattern because of the smaller size. We use PKC to refer to our algorithm with this optimization.

We observe that there could still be load imbalance in our algorithm, because a simple partitioning scheme is used to assign vertices to the threads in the scanning phase. We will consider techniques to decrease workload imbalance in our future work.

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our method PKC (the code is available at https://github.com/humayunk1/PKC), and compare our method to ParK [16], BZ [10], MPM [11] and an

implementation of k-core decomposition in the Ligra [17] graph processing library. We start by describing the evaluation approach, followed by comparison of serial performance of all the algorithms. Next, we compare PKC, ParK [16], and MPM [11] in terms of scalability and efficiency. Finally, we replace the k-core decomposition routine in the maximum clique finding software pmc [6] and report the speedup of the overall approach.

A. Experimental Setup

We describe the experimental setup in this subsection, including the system used, the test graph suite, and measurement metrics used for comparison.

System and Software. We test the methods on a shared-memory server with 32 cores. This server contains four Intel Xeon E7-8837 processors. Each processor has eight cores and supports one hardware thread. The node also has 512 GB main memory.

All the codes are compiled using the Intel C compiler (version 15.0.0) with -O3 optimization. We use OpenMP for parallelization, and threads are pinned to cores using the *compact* pinning strategy.

Test Suite. We choose several large undirected graphs to evaluate our method. The test suite consists of 17 large sparse graphs. These graphs are collected from the University of Florida Sparse Matrix collection [18] and the Stanford Network Analysis Project [19]. The graphs are listed in Table I. Each row of the table contains a graph name, number of vertices and edges in the graph, and maximum degree and coreness value for any vertex in the graph. The number of vertices varies from 1.14 to 118.14 million, and number of edges varies from 11.10 to 1949 million. The largest graph in terms of edges contains 1949 million edges and the largest graph in terms of vertices contains 118.14 million vertices.

Performance Metric. The main performance metric we consider to compare the methods is execution time. We run each method multiple times (the iteration count varies with thread concurrency) and report the average execution time in seconds.

B. Performance

We compare the execution time and speedup of our method PKC with ParK, BZ, and a shared-memory implementation of MPM. In PKC, we empirically chose f to be 0.98. However, this value may not be optimal for all graphs. In future work, we will investigate alternate strategies to pick the crossover point. We start with comparing performance of serial execution time of the methods.

Serial Performance. The Batagelj and Zaversnik (BZ) algorithm is a linear-time $O(n + m)$ algorithm for computing k-core decomposition of a graph. Because of this time complexity, BZ algorithm is one of the most popular methods to compute k-core decomposition. To compare different methods against BZ, we define speedup relative to BZ. If $bz\_t(g)$ is the execution time of BZ algorithm for a graph g and $method\_t(g)$ is the execution time of a $method$ (PKC, ParK, MPM) for the same graph g, then we define the speedup relative to BZ as the following ratio:

$$\frac{bz\_t(g)}{method\_t(g)}.$$

We compare the performance of four sequential methods in Figure 3. We observe that PKC performs the best among the methods. PKC achieves a significant speedup for most of the graphs. The second best method is BZ. BZ outperforms MPM for all the graphs and also performs better than ParK

Table I
THE TEST SUITE OF GRAPHS USED IN OUR STUDY. THE NUMBER OF VERTICES (n), NUMBER OF EDGES (m), MAXIMUM DEGREE (d_{MAX}), AND MAXIMUM CORE VALUE (K_{MAX}) ARE GIVEN.

Graph	n ($\times 10^6$)	m ($\times 10^6$)	$d_{\max}$	$K_{\max}$
as-skitter	1.70	11.10	35 455	111
cit-Patents	3.77	16.52	793	64
in-2004	1.38	16.92	21 869	488
soc-pokec	1.63	30.62	14 854	47
wb-edu	9.85	57.16	25 781	448
soc-LiveJournal1	4.85	68.99	20 333	372
ljournal-2008	5.36	79.02	19 432	425
hollywood-2009	1.14	113.89	11 467	2208
com-orkut	3.07	117.19	33 313	253
indochina-2004	7.41	194.11	256 425	6869
uk-2002	18.52	298.11	194 955	943
arabic-2005	22.74	640.00	575 628	3247
uk-2005	39.46	936.36	1 776 858	588
webbase-2001	118.14	1019.90	816 127	1506
it-2004	41.29	1150.73	1 326 744	3224
soc-friendster	65.61	1806.07	5214	304
sk-2005	50.64	1949.41	8 563 816	4510

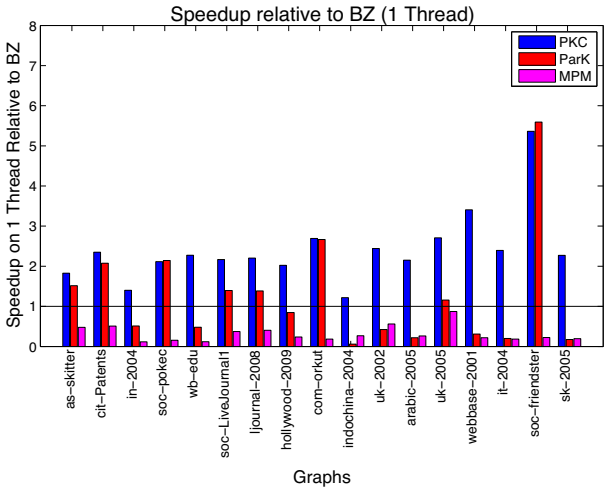

Figure 3. Serial speedup of different methods relative to BZ. PKC performs better than all the other methods. However, BZ performs better than MPM for all the graphs and also outperforms ParK for 9 graphs. The geometric mean speedups of PKC, ParK, and MPM on the entire test suite are 2.29, 0.71, and 0.27, respectively.

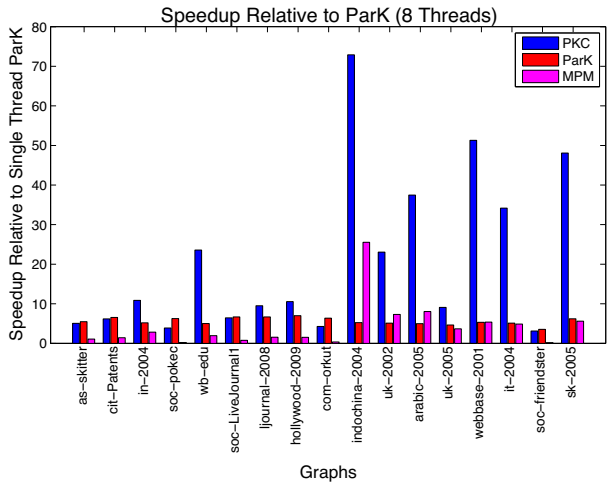

Figure 4. Parallel speedup achieved by ParK, PKC, and MPM when using 8 threads. Speedup is relative to ParK performance on a single thread. The geometric mean of the speedups of the graphs in Table I for PKC, ParK and MPM are 13.31, 5.54, and 2.04 respectively.

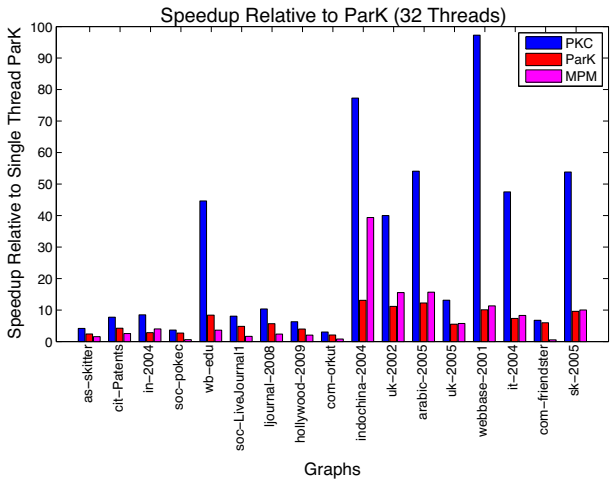

Figure 5. Parallel speedup achieved by ParK, PKC, and MPM when using 32 threads. Speedup is relative to ParK performance on a single thread. The geometric mean speedups of PKC, ParK, and MPM on the entire test suite are 15.89, 5.65, and 3.76, respectively.

for most of the graphs. The overall speedups (geometric mean) of PKC, ParK and MPM in comparison to BZ on the entire test suite are 2.29, 0.71, and 0.27, respectively. Unsurprisingly, MPM is slower as it performs significantly more work than BZ and ParK.

Parallel Performance. To compare the parallel performance and scalability of different methods, we determine the speedup of the methods relative to single-threaded time of ParK. If ParK_$t(g,1)$ is the execution time of ParK with 1 thread for a graph g, and $method\_t(g,p)$ is the execution time of a $method$ (PKC, MPM) with p threads for the same graph g, then we define the speedup as:

$$\frac{\text{ParK\_}t(g,1)}{method\_t(g,p)}.$$

We first report the speedups with 8 threads in Figure 4. We observe that PKC performs better than both ParK and MPM for most of the graphs. PKC achieves significant speedup for graphs from LAW group. For example, indochina-2004 achieves a speedup of 73. The value of $K_{\max}$ for the LAW graphs is relatively high. Most of the vertices belong to lower k-core and most of the higher k-core subgraphs are empty. PKC exploits this observation, by considering only a fraction of vertices. This decreases the SCAN time and processing time of the high degree vertices significantly. Additionally, ParK makes more barrier synchronization calls than PKC and also uses more atomic operations. For these reasons, PKC achieves the best performance among the methods. We expect that performance difference of the methods would be more pronounced on higher core count, as the cost of barriers and atomic operations would be higher on higher core count. The MPM method is not competitive because of the redundant work performed.

Table II
EXECUTION TIME OF PKC, PARK AND MPM ON 32 THREADS FOR THE WHOLE TEST SUITE.

Graph	Execution time (s)		
	PKC	ParK	MPM
as-skitter	0.23	0.39	0.60
cit-Patents	0.30	0.55	0.92
in-2004	0.22	0.65	0.46
soc-pokec	0.40	0.54	2.39
wb-edu	0.32	1.72	3.98
soc-LiveJournal1	0.87	1.44	4.20
ljournal-2008	0.81	1.47	3.55
hollywood-2009	1.12	1.77	3.44
com-orkut	2.38	3.49	9.22
indochina-2004	1.77	10.45	3.47
uk-2002	1.42	5.07	3.65
arabic-2005	4.06	17.93	14.01
uk-2005	6.64	15.81	15.21
webbase-2001	5.47	52.70	46.95
it-2004	8.35	53.94	47.82
soc-friendster	31.32	35.51	386.74
sk-2005	15.44	86.60	83.10

We compare the methods ParK, PKC and MPM using 32 threads on the shared-memory node. The speedup of the methods relative to ParK on 1 thread is shown in Figure 5. Our method PKC performs better than both ParK and MPM. We observe that the performance difference of the methods is more pronounced with a larger thread count. This is because ParK method uses more OpenMP barriers and atomic operations as compared to PKC. Both these operations have higher cost on 32 threads as compared to 8 threads. Also, the SCAN operation and processing of vertices

takes less time for PKC. The performance difference is particularly noticeable for the LAW graphs. We note that PKC performs best for the graph webbase-2001 (speedup 97). Overall PKC achieves a speedup of 15.89 for the whole test suite, whereas the overall speedup of ParK and MPM are 5.65 and 3.76, respectively.

To understand the performance of PKC better, we count the number of adjacencies (edges) visited by each thread. This corresponds to the work performed by each thread. We compute the ratio between the slowest thread (maximum number of edges visited by a thread) and the average number of edges visited by a thread in 32 threaded execution. We observe that this ratio varies from 1.15 to 6.92 and the geometric mean of the ratios is 1.94. Thus there is a noticeable workload imbalance for some graphs. This is because we use a simple static scheduling technique to assign the vertices to the threads. We also considered dynamic scheduling approach to partition the vertices among the threads, however this takes more time than static scheduling. So we use static scheduling in this work. We note that there is room for further improvement and sophisticated scheduling strategy could be used to decrease workload imbalance among the threads.

We report the execution time of the methods PKC, ParK, and MPM in Table II. The execution time of PKC is less than that of ParK. This difference in execution time is more noticeable for bigger graphs. The bigger graphs take more time to compute k-core decomposition. This decrease in execution time would allow us to process even bigger graphs. It can be observed that MPM takes a lot more time for the bigger graphs.

We next consider aggregate performance of the methods on 1-32 threads for the whole test suite in Table I. We define the speedup for a method relative to the serial execution time of ParK. The aggregate speedup is defined as the geometric mean of the speedups of the graphs in Table I. If the execution time of ParK for a graph g on 1 thread is ParK_$t(g,1)$ and the execution time of a $method$ (ParK, PKC, MPM) for a graph g on p threads is $method\_t(g,p)$, the aggregate speedup on p threads for a $method$ (ParK, PKC and MPM) is defined as follows:

$$geomean(\frac{\text{ParK\_}t(g,1)}{method\_t(g,p)}), s.t., g \in Table\ I.$$

The aggregate speedup of the methods is shown in Figure 6. We observe that PKC performs consistently better than both ParK and MPM on 1-32 cores. The barrier and atomic operations overhead is not that much on one processor (1-8 cores), however PKC performs better because of decreasing time in the scan and processing phase. On multiple processors (8-32 cores), the barrier synchronization cost and atomic operation cost increases. This increase in cost of these operations, makes the difference in performance

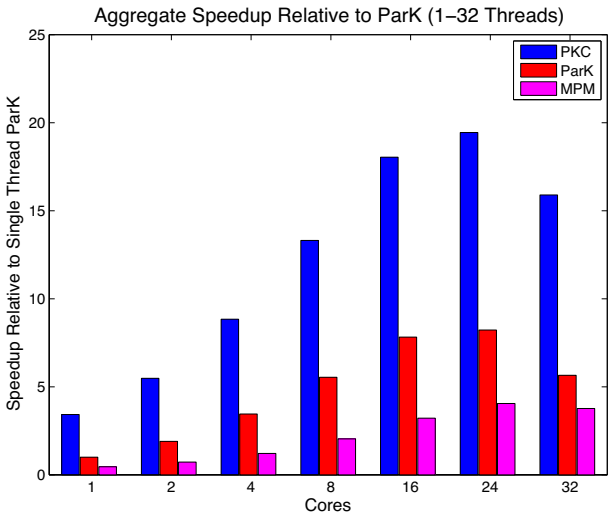

Figure 6. Aggregate speedup of PKC, ParK and MPM for all the graphs in Table I on 1-32 cores. We observe that PKC is consistently scaling better than ParK and MPM, and the speedup of PKC is increasing with increasing core count.

among the methods more prominent. PKC scales consistently better than the other two methods and it performs the best on 24 threads with a geometric mean of speedups of 19.44. On 32 threads, our method performs better than the other methods. However, there is a decrease in speedups for all the methods on 32 threads from 24 threads. This could be because of workload imbalance among the threads. There are only few vertices in the higher cores, thus it is possible that some threads are idle. We would investigate this further in our future work. This performance improvement by PKC over the other methods, makes our approach relevant on shared-memory systems.

We analyze the graph soc-LiveJournal1 to see if the execution time and the number of levels are correlated. The number of levels and the number of sub-levels present in each level for this graph are shown in Figure 7. It can be seen that in some levels, there are a high number of sub-levels, and thus there could be a considerable number of barrier calls, thus increasing execution time for that level. This is observed in Figure 8. The execution time for a level is expected to be high if the number of sub-levels in that level is high. Also, the execution time of ParK is higher for those levels. The execution time of PKC is also related to number of sub-levels. However, the time taken by PKC is not that sensitive to the number of sub-levels, thus taking less time overall. The peak in time for PKC at level 50 is from the creation of a smaller array containing the rest of the vertices and making the graph with remaining vertices. PKC performs very well after level 50. This is because PKC is scanning a smaller array and processing the vertices using

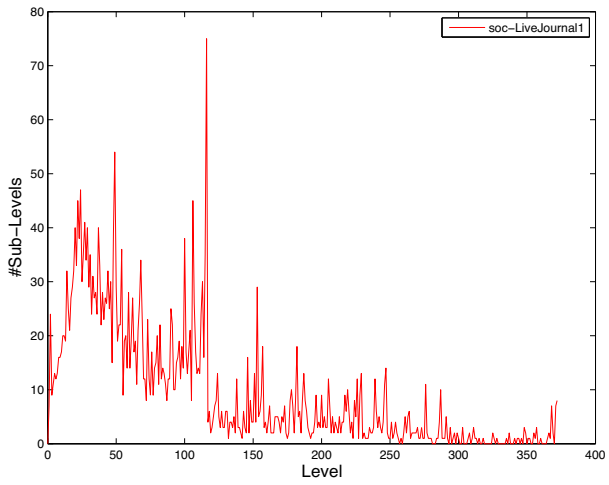

Figure 7. The number of levels and the number of sub-levels in each level for the ParK execution on the soc-LiveJournal1 graph.

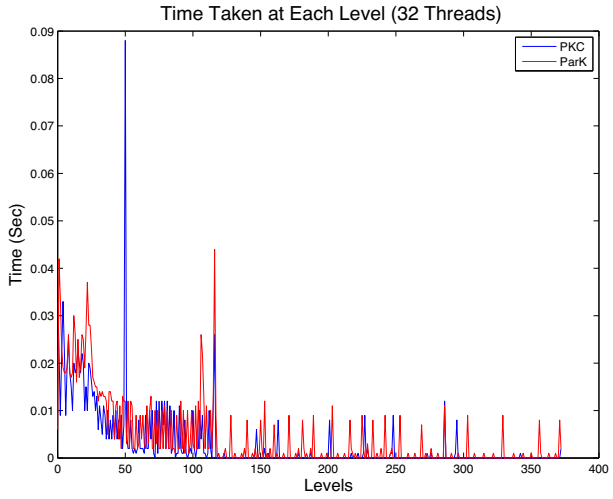

Figure 8. The time taken by ParK and PKC for each level on the soc-LiveJournal1 graph. We observe that time taken in a level is related to the number of sub-levels present in the level.

a smaller graph.

We also compare the performance of ParK, PKC, MPM and the k-core implementation in Ligra. We run Ligra for five graphs on 32 threads. We find that Ligra takes two to three orders of magnitude more time compared to the other methods. We decided to omit the performance results of Ligra for this reason.

Now, we consider an application of k-core decomposition. k-core decomposition is used in the pmc algorithm to find the maximum clique in a graph [6]. The maximum clique finding problem is NP-hard. A heuristic approach is to use k-core decomposition as a first step. The k-core of a graph,

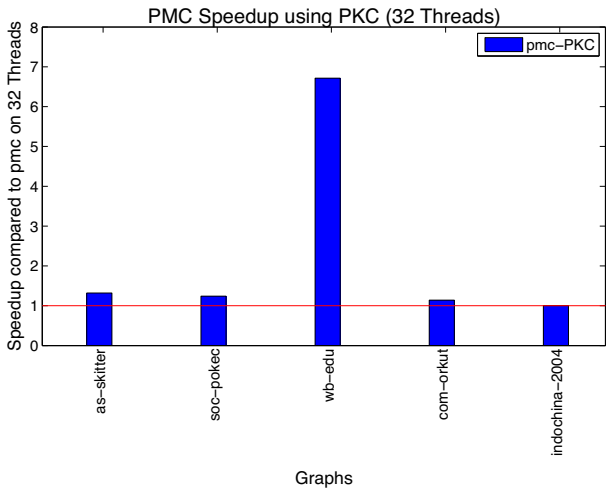

Figure 9. The speedup achieved by the pmc maximum clique finding method when using PKC for k-core decomposition. The overall performance of pmc improves, and for some graphs, the speedup is more than $6.71\times$.

which is defined as the maximum k-core vertex present in a graph, gives an upper bound on the size of the maximum clique present in a graph. The maximum clique is found using a branch and bound algorithm (Bron-Kerbosch) [20]. The k-core values of the vertices provides an upper bound on the size of maximum clique that a given vertex can be part of. In pmc [6], to find maximum clique in parallel, the BZ [10] algorithm is used for k-core decomposition of a graph. We use PKC to compute k-core decomposition in pmc instead. The speedup of pmc when PKC is used to compute the k-core decomposition is shown in Figure 9. The BZ algorithm takes significant amount of time in pmc, thus speeding up k-core decomposition improves overall performance of pmc. We observe a $6.71\times$ speedup for some graphs. For example, on wb-edu, k-core decomposition takes more than 89% of the total time in the original approach. Thus, speeding up the k-core computations gives a noticeable speedup.

V. Conclusions and Future Work

We presented a new algorithm PKC for k-core decomposition. The algorithm is simpler than the state-of-the-art ParK algorithm, uses less atomic operations, has lower synchronization overhead, and decreases scan and processing time by considering a smaller graph. We discussed the algorithm and its OpenMP implementation in detail. We showed that PKC achieves a $2.81\times$ speedup compared to ParK on a suite of large test networks on 32 threads. We also showed that it is faster than other parallel k-core implementations.

In future work, we will investigate further improvements to this algorithm. The algorithm uses a very simple work partitioning scheme which works reasonably well for 32

threads. However, for larger thread concurrencies, a more sophisticated work partitioning scheme might be required. Another line of future work is to develop implementations of k-core decomposition for manycore platforms such as Intel Xeon Phi processors and GPUs.

ACKNOWLEDGMENTS

This work is supported by the US National Science Foundation grants ACI-1253881 and CCF-1439057.

REFERENCES

[1] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, pp. 269–287, 1983.

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

[3] 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.

[4] Y. Cheng, C. Lu, and N. Wang, "Local k-core clustering for gene networks," in *Proc. IEEE Int'l. Conf. on Bioinformatics and Biomedicine (BIBM)*, 2013.

[5] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: a tool for the analysis of large scale internet graphs," arXiv.org e-Print archive, http://arxiv.org/abs/cs/0511007, 2005.

[6] 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.

[7] C. Peng, T. G. Kolda, and A. Pinar, "Accelerating community detection by using k-core subgraphs," arXiv.org e-Print archive, https://arxiv.org/abs/1403.2226, 2014.

[8] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "D-cores: measuring collaboration of directed graphs based on degeneracy," *Knowledge and Information Systems*, vol. 35, pp. 311–343, 2013.

[9] A. Garas, F. Schweitzer, and S. Havlin, "A k-shell decomposition method for weighted networks," *New Journal of Physics* vol. 14, no. 8, p. 083030, Aug. 2012.

,

[10] 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.

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

[12] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single PC," *Proc. VLDB Endow.*, vol. 9, no. 1, pp. 13–23, 2015.

[13] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2012.

[14] P. Boldi and S. Vigna, "The Webgraph Framework I: Compression techniques," in *Proc. Int'l. Conf. on World Wide Web (WWW)*, 2004.

[15] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu, "Efficient core decomposition in massive networks," in *Proc. IEEE Int'l. Conf. on Data Engineering (ICDE)*, 2011.

[16] N. S. Dasari, D. Ranjan, and M. Zubair, "ParK: An efficient algorithm for k-core decomposition on multicore processors," in *Proc. Int'l. Workshop on High Performance Big Graph Data Management, Analysis, and Mining (BigGraphs)*, 2014.

[17] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[18] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, pp. 1:1–1:25, 2011, http://www.cise.ufl.edu/research/sparse/matrices.

[19] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[20] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, 1973.