View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Parallel and I/O Efficient Set Covering Algorithms

Guy E. Blelloch Harsha Vardhan Simhadri Kanat Tangwongsan
Carnegie Mellon University
{guyb, hsimhadr, ktangwon}@cs.cmu.edu

ABSTRACT

This paper presents the design, analysis, and implementation of parallel and sequential I/O-efficient algorithms for set cover, tying together the line of work on parallel set cover and the line of work on efficient set cover algorithms for large, disk-resident instances.

Our contributions are twofold: First, we design and analyze a parallel cache-oblivious set-cover algorithm that offers essentially the same approximation guarantees as the standard greedy algorithm, which has the optimal approximation. Our algorithm is the first efficient external-memory or cache-oblivious algorithm for when neither the sets nor the elements fit in memory, leading to I/O cost (cache complexity) equivalent to sorting in the Cache Oblivious or Parallel Cache Oblivious models. The algorithm also implies low cache misses on parallel hierarchical memories (again, equivalent to sorting). Second, building on this theory, we engineer variants of the theoretical algorithm optimized for different hardware setups. We provide experimental evaluation showing substantial speedups over existing algorithms without compromising the solution's quality.

Categories and Subject Descriptors: F.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity

General Terms: Algorithms, Theory

Keywords: Parallel algorithms, set cover, max k-cover, external memory algorithms, approximation algorithms.

1. INTRODUCTION

Set cover is one of the most fundamental and well-studied problems in optimization and approximation algorithms. For decades, this problem and its variants have found many applications, including locating warehouses, testing faults, scheduling crews on airlines, and allocating wavelength in wireless communication. These applications often have to deal with massive data [14] and are well-suited for parallel and I/O efficient algorithms.

Let $\mathcal{U}$ be a set of n ground elements, $\mathcal{F}$ be a collection of subsets of $\mathcal{U}$ that together covers $\mathcal{U}$ (i.e., $\cup_{S\in\mathcal{F}} S = \mathcal{U}$), and $c\colon \mathcal{F} \to \mathbb{R}_+$ be a cost function. The *set cover problem* is to find the cheapest collection of sets $\mathcal{A} \subseteq \mathcal{F}$ that that covers $\mathcal{U}$ (i.e., $\cup_{S\in\mathcal{A}} S = \mathcal{U}$), where the cost of a solution $\mathcal{A}$ is specified by $c(\mathcal{A}) = \sum_{S\in\mathcal{A}} c(S)$. Unweighted set cover (all weights are equal) appeared as one of the 21 problems Karp identified as NP-complete in 1972 [21]. Two years later, Johnson [19] proved that the simple greedy method gives an approximation that is at most a factor $H_n = \sum_{k=1}^{n} \frac{1}{k}$ from optimal. Subsequently, Chvátal [13] proved the same approximation bounds for the weighted case. These results are complemented by a matching hardness result: Feige [16] showed that unless $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{O(\log\log n)})$, set cover cannot be approximated in polynomial time with a ratio better than $(1 - o(1)) \ln n$. This essentially shows that the greedy algorithm is optimal. Furthermore, the greedy algorithm is particularly simple running in $O(W)$ time for the unweighted case and $O(W \log W)$ time for the weighted case. Here, $W \geq n$ is the sum of the sizes of the sets. In the bipartite graph view we will use later on, the quantity W is the number of edges in the graph.

The greedy algorithm seems hard to parallelize directly, but Berger, Rompel, and Shor [2] (BRS) showed that it can be "approximately" parallelized leading to an $O(\log^5 W)$-depth and $O(W \log^4 W)$-work randomized algorithm, giving a $(1 + \varepsilon)H_n$-approximation on a PRAM. Rajagopalan and Vazirani [24] later gave an improvement in both work and depth. More recently, this was improved to linear work and smaller depth [7]. In terms of I/O efficient algorithms, Cormode, Karloff, and Wirth recently developed an efficient algorithm for the case when the elements—but not necessarily the sets—fit in memory [14]. We know of *no* I/O efficient solutions for the general case (i.e. when neither the sets nor the elements fit in memory). Furthermore, the CKW algorithm is strictly sequential.

Our Contributions: This paper presents the design, analysis, and experimental evaluation of set-covering algorithms. First, we design and analyze approximation algorithms for set cover and related problems that are *both parallel and I/O efficient*. These are the first results we know of for I/O efficient set cover where neither the sets nor elements fit in memory. Building on this theory, we implemented slight variants of the theoretical algorithm optimized for two different hardware setups: one a high-end parallel workstation and the other a "wimpy" machine with small memory but a fast disk. We provide extensive experimental evaluation show-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'12, June 25–27, 2012, Pittsburgh, Pennsylvania, USA.
Copyright 2012 ACM 978-1-4503-1213-4/12/06 ...$10.00.

ing non-trivial speedups over existing algorithms without compromising the solution's quality.

Our theoretical results show how to implement recent results for parallel set cover [7] (BPT) in an I/O efficient manner. In the cache oblivious model [17] with cache size parameter M and block size B, our algorithm has cache complexity $O(\frac{W}{B}\log_{\frac{M}{B}}\frac{W}{B})$. This matches the complexity of sorting. Furthermore, we achieve polylogarithmic depth and the same cache complexity bounds in the more stringent parallel cache oblivious model [4]. Our results modify the BPT algorithm with appropriate data structures. The algorithms give an $(1+\varepsilon)H_n$-approximation for arbitrary $\varepsilon > 0$ and hence are essentially optimal. In addition to set cover, as shown in [7], the same sequence of sets can be used as a solution to max cover and min-sum set cover. For max cover, this sequence is prefix optimal: for any prefix of length k, this prefix is a $(1-1/e-\varepsilon)$-approximation to the max k-cover problem.

We have implemented and experimented with two variants of our algorithm, one for shared-memory parallel machines and one for external memory. The main difference between the two variants is how we implement the communication steps. We experiment with the parallel variant on a modern 40-core machine and with the external memory variant on a more modest machine using a solid-state drive (SSD). We test the algorithms with several large instances with up to 5.5 billion edges. For the parallel version, we are able to achieve significant speedup over a fast sequential implementation. In particular, we compare to the CKW sequential algorithm which is already significantly faster than the greedy algorithm. For our largest graph with about 5.5 billion edges, the algorithm runs in around 10 seconds and is 13.5x faster than the CKW algorithm. In fact, it runs faster than optimized parallel sorting code [25] on the same sized data. For the max k-cover problem, we empirically show that it is often possible to speedup the computation by more than a factor of 2 by stopping the algorithm early when k is known and is small relative to the set cover's solution size. For the sequential I/O variant, we are able to achieve orders of magnitude speedups over the results of CKW when neither the sets nor elements fit in memory. When the elements fit in memory, the CKW algorithm is faster. With regards to quality of the results (number of sets returned), our algorithm returns about the same number of sets as the other algorithms.

2. PRELIMINARIES AND NOTATION

Let $[k]$ be the set $\{1, 2, \ldots, k\}$. For a graph G, we denote by $\deg_G(v)$ the degree of the vertex v in G. We denote by $N_G(v)$ the neighbor set of the node v and by $N_G(X)$ the neighbors of the vertex set X, i.e., $N_G(X) = \cup_{w\in X} N_G(w)$. We drop the subscript when the context is clear. Let $V(G)$ and $E(G)$ denote respectively the set of nodes and the set of edges. We also write $|G|$ to mean the number of edges of G. For an input of size W, we assume that every memory word has $O(\log W)$ bits.

Computation Model. We present algorithms in the nested parallel model, allowing arbitrary dynamic nesting of parallel loops and fork-join constructs but no other synchronizations. This corresponds to the class of algorithms with series-parallel dependence graphs (see Figure 1). Computations can be decomposed into "tasks", "parallel blocks" and "strands" recursively: As a base case, a ***strand*** is a serial sequence of instructions not containing any parallel constructs or subtasks. A ***task*** is formed by serially composing $k \geq 1$ strands interleaved with $(k-1)$ "parallel blocks" (denoted by $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \ldots; \mathsf{s}_k$). A ***parallel block*** is formed by composing in parallel one or more tasks with a fork point before all of them and a join point after (denoted by $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$). A parallel block can be, for example, a parallel loop or some constant number of recursive calls. The top-level computation is a task. The ***span*** (aka. ***depth***) of a computation is the length of the longest path in the dependence graph.

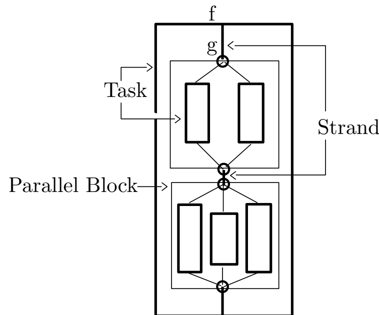
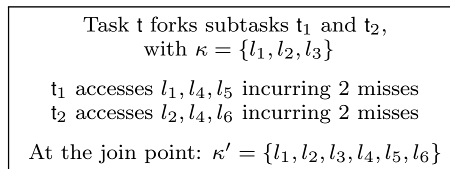


Figure 1. Decomposing the computation: tasks, strands and parallel blocks

Measuring Memory Access Costs. We will analyze algorithms in the Parallel Cache Oblvivious (PCO) model [4], which is a parallel variant of the cache oblivious (CO) model and gives cache complexity costs that are always at least as large. Therefore any upper bounds on the PCO are also upper bounds on the CO model. The Cache Oblivious (CO) model [17] is a model for measuring cache misses of an algorithm, when run on a single processor machine with a two-level memory hierarchy—one level of finite cache and a memory of unbounded size. The cache complexity measure of an algorithm under this model $Q(n; M, B)$ counts the number of cache misses incurred by a problem instance of size n when run on a fully associative cache of size M and line size B using the optimal offline algorithm (i.e., the optimal cache replacement policy). Throughout the paper we assume that $M \geq B$ (the *tall cache assumption*).

Like the cache-oblivious model, in the ***Parallel Cache-Oblivious (PCO) model***, there is a memory of unbounded size and a single cache with size M, line-size B (in words), and optimal replacement policy. The cache state κ consists of the set of cache lines resident in the cache at a given time. When a location in a non-resident line l is accessed and the cache is full, l replaces in κ the line accessed furthest into the future, incurring a *cache miss*. To extend the CO model to parallel computations, one needs to define how to analyze


Task t forks subtasks t_1 and t_2,
with $\kappa = \{l_1, l_2, l_3\}$

t_1 accesses l_1, l_4, l_5 incurring 2 misses
t_2 accesses l_2, l_4, l_6 incurring 2 misses

At the join point: $\kappa' = \{l_1, l_2, l_3, l_4, l_5, l_6\}$

Figure 2. Applying the PCO model (Definition 2.1) to a parallel block. Here, $Q^*(\mathsf{t}; M, B; \kappa) = 4$.

the number of cache misses for the tasks that run in parallel in a parallel block. The PCO model approaches it by (i) ignoring any data reuse among the parallel subtasks and (ii) assuming the cache is flushed at each fork and join point of any task that does not fit within the cache.

More formally, let $loc(\mathsf{t}; B)$ denote the set of distinct cache lines accessed by task t, and $S(\mathsf{t}; B) = |loc(\mathsf{t}; B)| \cdot B$ denote its size (also let $s(\mathsf{t}; B) = |loc(\mathsf{t}; B)|$ denote the size in terms of number of cache lines). Let $Q(\mathsf{c}; M, B; \kappa)$ be the cache complexity of c in the sequential CO model when starting with cache state κ.

Definition 2.1 (Parallel Cache-Oblivious Model) *For cache parameters M and B the **cache complexity** of a strand, parallel block, and a task starting in cache state κ are defined recursively as follows (refer [4] for more details).*

- *For a strand,* $Q^*(\mathsf{s}; M, B; \kappa) = Q(\mathsf{s}; M, B; \kappa)$.
- *For a parallel block* $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$,
  $Q^*(\mathsf{b}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{t}_i; M, B; \kappa)$.
- *For a task* $\mathsf{t} = \mathsf{c}_1; \ldots; \mathsf{c}_k$,
  $Q^*(\mathsf{t}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{c}_i; M, B; \kappa_{i-1})$,
  where $\kappa_i = \emptyset$ *if* $S(\mathsf{t}; B) > M$, *and* $\kappa_i = \kappa \cup_{j=1}^{i} loc(\mathsf{c}_j; B)$ *if* $S(\mathsf{t}; B) \le M$.

We use $Q^*(\mathsf{c}; M, B)$ to denote a computation c starting with an empty cache, $Q^*(n; M, B)$ when n is a parameter of the computation. We note that $Q^*(\mathsf{c}; M, B) \ge Q(\mathsf{c}; M, B)$. When applied to a parallel machine, Q^* is a "work-like" measure and represents the total number of cache misses across all processors. An appropriate scheduler is used to evenly balance them across the processors.

Primitives. We need primitives such as sorting, prefix sums, merge, filter, map for the set cover algorithm. Parallel algorithms with optimal cache complexity in the PCO model and polylogarithmic depth can be constructed for these problems (for details, see [5, 4]). The cache complexity of sorting on an input instance n in the PCO model is $\text{sort}(n; M, B) = O(\frac{n}{B} \log_{M/B} \frac{n}{B})$, while the complexity of the other primitives is $O(n/B)$. We use $\text{sort}(n)$ as shorthand. All these primitives have $O(\log^2 n)$ depth.

3. ALGORITHM DESIGN

This section describes an efficient implementation of the BPT set cover algorithm [7] in the PCO model, implying good I/O complexity in other related models. We begin by reviewing the BPT algorithm and describing how to achieve an I/O efficient algorithm that satisfies Theorem 3.1. In the following section, we discuss optimizations we made to the theoretical algorithm to achieve good performance on different hardware setups.

We state the complexity of the set cover algorithm in terms of W, which, as defined previously, is the sum of the set sizes:

Theorem 3.1 (Parallel and I/O Efficient Set Cover) *The I/O (cache) complexity of the randomized approximate set cover algorithm on an instance of size W is expected $O(sort(W))$ and the depth is $O(\text{polylog}(W))$ **whp.** Furthermore, this implies an algorithm for prefix-optimal max cover and min-sum set cover in the same complexity bounds.*

Consider the BPT algorithm in Algorithm 3.1. At the core of it are the following 3 ingredients—prebucketing, maximal near-independent set (MaNIS), and bucket management—which we discuss in turn:

Algorithm 3.1 `SetCover` — Blelloch et al. parallel greedy set cover.

Input: a set cover instance $(\mathcal{U}, \mathcal{F}, c)$ and a parameter $\varepsilon > 0$.
Output: a *ordered* collection of sets covering the ground elements.

i. Let $\gamma = \max_{e \in \mathcal{U}} \min_{S \in \mathcal{F}} c(S)$, $W = \sum_{S \in \mathcal{F}} |S|$, $T = \log_{1/(1-\varepsilon)}(W^3/\varepsilon)$, and $\beta = \frac{W^2}{\varepsilon \cdot \gamma}$.
ii. Let $(\mathcal{A}; A_0, \ldots, A_T) = \texttt{Prebucket}(\mathcal{U}, \mathcal{F}, c)$ and $\mathcal{U}_0 = \mathcal{U} \setminus (\cup_{S \in \mathcal{A}} S)$.
iii. For $t = 0, \ldots, T$, perform the following steps:
  1. Remove deleted elements from sets in this bucket: $A'_t = \{S \cap \mathcal{U}_t : S \in A_t\}$
  2. Only keep sets that still belong in this bucket: $A''_t = \{S \in A'_t : c(S)/|S| > \beta \cdot (1-\varepsilon)^{t+1}\}$.
  3. Select a maximal nearly independent set from the bucket: $J_t = \texttt{MaNIS}_{(\varepsilon, 3\varepsilon)}(A''_t)$.
  4. Remove elements covered by J_t: $\mathcal{U}_{t+1} = \mathcal{U}_t \setminus X_t$ where $X_t = \cup_{S \in J_t} S$
  5. Move remaining sets to the next bucket: $A_{t+1} = A_{t+1} \cup (A'_t \setminus J_t)$
iv. Finally, return $\mathcal{A} \cup J_0 \cup \cdots \cup J_T$.

— *Prebucketing:* This component (Step ii of the algorithm) buckets the sets based on their cost. To ensure that the ratio between the costliest set and cheapest set is polynomially bounded so that the total number of buckets is kept logarithmic, as described in Lemma 4.2 of [7], sets that cost more than a threshold are discarded, and all sets cheaper than a certain threshold ($\mathcal{A}$) are included in the solution and the elements in these included sets marked as covered. Then $\mathcal{U}_0$ consists of the uncovered elements. The remaining sets are placed into $O(\log W)$ buckets $(A_0, A_1, \ldots, A_T)$ by their normalized cost (cost per element).

The algorithm then enters the main loop (step iii.), iterating over the buckets from the least to the most expensive and invoking MaNIS once in each iteration.

— MaNIS*:* Invoked in Step iii(3) of the set cover algorithm, MaNIS finds a subcollection of the sets in a bucket that are almost non-overlapping with the goal of closely mimicking the greedy behavior. Algorithm 3.2 shows the MaNIS algorithm, reproduced from [7]. (The annotations on the side indicate which primitives in the PCO model we will use to implement them.) Conceptually, the input to MaNIS is a bipartite graph with left vertices representing the sets and the right vertices representing the elements. The procedure starts with each left vertex picking random priorities (step 2). Then, each element identifies itself with the highest priority set containing it (step 3). If "enough" elements identify themselves with a set, the set selects itself (step 4). All selected sets and the elements they cover are eliminated (steps 5(1), 5(2)), and the cost of remaining sets is re-evaluated based on the uncovered elements. Only sets (A') with costs low enough to belong to the correct bucket (which invoked this MaNIS) are selected in step 5(3) and the procedure continues with another level of recursion in step 6. The net result is that we choose nearly non-overlapping sets, and the sets that are not chosen are "shrunk" by a constant factor.

— *Bucket Movement:* The remaining steps in Algorithm 3.1 are devoted to moving sets between buckets, ensuring the contents of the least-costly bucket contain only sets in a

specific cost range. Step iii(4) removes elements covered by the sets returned by MaNIS; step iii(5) transfers the sets not selected in MaNIS to the next bucket; and step iii(2) selects only those sets with costs in the current bucket's range for passing to MaNIS.

3.1 I/O Efficient Algorithm

We now discuss the right set of data structures and primitives to make the above algorithm both parallel and I/O efficient. In both the set cover and MaNIS algorithms, the set-element instance is represented as a bipartite graph with sets on the left. A list of sets as well as a compact and contiguous adjacency list for each set is stored. The universe of elements $\mathcal{U}$ is represented as a bitmap indexed by an element identifier which is updated to indicate when element has been covered. Since we only need one bit of information per element to indicate whether it is covered or not, this can be stored in $O(\frac{|\mathcal{U}|}{\log W})$ words. Unlike in [6], we do not maintain back pointers from the elements to the sets.

— *Prebucketing:* This phase involves sorting sets based on their cost and a filter to remove the costliest and the cheapest set. Sets can then be partitioned into buckets with a merge operation. All operations have less than sort(W) I/O complexity and $O(\log^2 n)$ depth in the PCO model.

— MaNIS*:* We invoke MaNIS in step 3 of the set cover algorithm. Inside MaNIS, we store the remaining elements of $\mathcal{U}$ (right vertices) as a sequence of element identifiers. To implement MaNIS, in Step 3, for each left vertex, we copy its value x_a to all the edges incident on it, then sort the edges based on the right vertex so that the edges incident on the same right vertex are contiguous. For each right vertex, we can now use a prefix "sum" using maximum to find the neighbor a with the maximum x_a. In step 4, the "winning" edges (an edge connecting a right vertex with it's chosen left vertex) are marked on the edge list for each right vertex we computed in step 3. The edge list is then sorted based on the left vertex. Prefix sum can then be used to compute the number of elements each set has "won". A compact representation for J and its adjacency list can be computed with a filter operation. In step 5(1), the combined adjacency list of elements in J is sorted and duplicates removed to get $\overline{B}$. For step 5(2), we first evaluate the list $A \setminus J$. Then, we sort the edges incident on $A \setminus J$ based on their right vertices; merge with the remaining elements to identify which is contained in $\overline{B}$, marking these edges accordingly. After sorting these edges based on their left vertices, for each left vertex $a \in A \setminus J$, we filter and pack the "live edges" to compute $N'_G(a)$. Steps 5(3) and 5(4) involve simple filter and sum operations. The most I/O intensive operation (as well as the operation with maximum depth) in each round of MaNIS is sorting, which requires at most $O(\text{sort}(|G|))$ I/O complexity and $O(\log^2 |G|)$ depth in the PCO model. As analyzed in [7], for a bucket with W_t edges to start with, MaNIS runs for at most $O(\log W_t)$ rounds—and after each round, the number of edges drops by a constant factor; therefore, we have the following bounds:

Lemma 3.2 *The cache (I/O) complexity in the PCO model of running* MaNIS *on a bucket with* W_t *edges is* $O(sort(W_t))$*, and the depth is* $O(D_{sort}(W_t) \log W_t)$*.*

— *Bucket Movement:* We assume the A_t, A'_t and A''_t are stored in the same format as the input for MaNIS (see 3.1). The right set of vertices of the bipartite graph is now a bitmap corresponding to the elements indicating whether an element is alive or dead. Step iii(1) is similar to Step 3 of MaNIS. We first sort $S \in A_t$ to order the edges by element index, then merge this representation (with a vector of length $O(|\mathcal{U}|/\log W)$) to match them with the elements bitmap, do a filter to remove deleted edges, and perform another sort to get them back ordered by set. Step iii(2) is simply a filter. The append operation in Step iii.5 is no more expensive than a scan. In the PCO models, these primitives have I/O complexity at most $O(\text{sort}(W_t) + \text{scan}(|\mathcal{U}|/\log W))$ for a bucket with W_t edges. They all have $O(D_{sort}(W))$ depth.

To show the final cache (I/O) complexity bounds, we make use of the following claim:

Claim 3.3 **([7])** *Let* W_t *be the number of edges in bucket* t *at the beginning of the iteration which processes this bucket. Then,* $\sum W_t = O(W)$*.*

Therefore, we have $O(\text{sort}(W))$ from prebucketing, $O(\text{sort}(W))$ from MaNIS combined, and $O(\text{sort}(W) + \text{scan}(\mathcal{U}))$ from bucket management combined (since there are $\log(W)$ rounds). This simplifies to an I/O (cache) complexity of $Q^*(W; M, B) = O(\text{sort}(W; M, B))$ since $U \leq W$. The depth is $O(\log^4 W)$, since set cover has $O(\log W)$ iterations to go through buckets, each and invoking a MaNIS which has $O(\log W)$ rounds (w.h.p.) and each round is dominated by the sort that has a maximum depth of $O(\log^2 W)$. of recursion.

4. IMPLEMENTATION

We highlight a number of design decisions that we made for the two versions of the MaNIS-based set cover algorithm: one optimized for the multicore architecture and the other for the external-memory setting. In both implementations, we represent the set system (i.e., the sets and their elements) as a contiguous array of integers listing the elements belonging to the sets; we also keep a pointer to the starting point of each set. This is essentially the compressed sparse row (CSR) format for sparse matrices. Furthermore, in both implementations, we maintain a "bitmap" vector that indicates whether or not an element has been covered by a set already; however, as detailed below, how we keep this vector depends on the particular implementation.

Parallel Implementation. This implementation targets modern machines with many cores and sufficient RAM to fit and process the dataset if sufficient care is taken to manage the memory. The goal of this implementation is therefore to take advantage of available parallelism and locality, and strike a balance between the computation cost and memory-access cost. We chose to implement the "bitmap" as a vector of integers of length $|\mathcal{U}|$. On the surface, this may seem like a waste of space, but we made this decision so that MaNIS can be efficiently implemented in-place using priority writes. Initially, we apply bucket sort and standard prefix computations to classify the input sets into the buckets they belong. To implement MaNIS, we notice that each round of MaNIS involves two major phases: (1) deciding for each element the "winning" set—the set with the highest priority that covers it and (2) subsequently, counting for each set that the elements on which it has won. The former phase can be done using concurrent priority writes, and the latter

Algorithm 3.2 $\texttt{MaNIS}_{(\varepsilon,3\varepsilon)}(G)$

Input: A bipartite graph $G = (A, N_G(a))$
A is a sequence of left vertices (the sets), and $N_G(a), a \in A$ are the neighbors of each left vertex on the right.
These are represented as contiguous arrays. The right vertices are represented implicitly as $B = N_G(A)$.
Output: $J \subseteq A$ of chosen sets.

1. If A is empty, return the empty set.
2. For $a \in A$, randomly pick $x_a \in_R \{0, \ldots, |G|^7 - 1\}$. *//map*
3. For $b \in B$, let φ be b's neighbor with maximum x_a *// sort and prefix sum*
4. Pick vertices of A "chosen" by sufficiently many in B: *// sort, prefix sum, sort and filter*

$$J = \{a \in A | \#\{b : \varphi(b) = a\} \geq (1 - 4\varepsilon)D(a)\}.$$

5. Update the graph by removing J and its neighbors, and elements of A with too few remaining neighbors:
   (1) $\overline{B} = N_G(J)$ (elements to remove) *// sort*
   (2) $N'_G = \{\{b \in N_G(a) | b \notin \overline{B}\} : a \in A \setminus J\}$ *// sort, merge, sort and filter*
   (3) $A' = \{a \in A \setminus J : |N'_G(a)| \geq (1 - \varepsilon)D(a)\}$ *// filter*
   (4) $N''_G = \{\{b \in N'_G(a)\} : a \in A'\}$ *// filter*
6. Recurse on reduced graph: $J_R = \texttt{MaNIS}_{(\varepsilon,3\varepsilon)}((A', N''_G))$
7. return $J \cup J_R$

phase involves random-accesses to the "bitmap" vector and resetting the values in the bitmap as necessary.

Our experiments show that simulating priority writes using compare and swap (CAS) on these sets in a standard way does not produce high contention—and is in fact faster than running sort a few times (like what was described previously). By doing so, we are able to implement MaNIS in-place, which helps reduce the memory footprint of our implementation. For performance, we also made efforts to minimize the number of passes over the bitmap and the data.

Extra care is given to how the MaNIS steps are implemented. To reduce the number of times a set needs to be processed within MaNIS, we use an approach which only processes the higher priority sets on each parallel round, using ideas similar to what is described in [3]. More specifically, we pre-order the sets by the (psuedo)-random priorities, and, on each round, process a prefix of this ordering. Since the prefix has high priorities, the sets are less likely to be forced to another round saving some work. In the limit, the prefix size would be one, effectively giving the CKW algorithm. In our experiments, we use a prefix containing about one fourth the original number of sets for all of the buckets except for the buckets with the largest sets. For the buckets with large sets, we specialize MaNIS to run with a prefix of size one (we term this `serialManis`). This leads to a light-weight implementation which can still take advantange of parallelism on the edges without the bulkiness of the full-fledged MaNIS.

Disk-optimized Implementation. At the other end of the spectrum, we target a single-core machine with so little fast memory that not even the bitmap—the bit indicator array for the elements—can fit in main memory, but this machine has relatively fast disk (e.g., a solid-state drive). In this case, random-accessing an array is in general very costly. Our implementation in this case follows the theoretical description rather closely: The "bitmap" vector is kept as a bit array—this most-obvious optimization yields substantially better performance than the alternative of using 1 byte per element. Initially, like in the parallel case, we apply bucket sort and standard prefix computations to classify the input sets into the buckets they belong. Each round of MaNIS, then, is implemented as a series of external-memory sorts and scans. We resort to STXXL, a C++ reimplementation of the Standard Template Library (STL) to perform external-memory (out-of-core) computations and disk-resident data management [15, 25]. STXXL hides from the users the intricate optimizations done at the low-level (e.g., asynchronous/bulk I/O).

5. EVALUATION

We empirically investigate the performance of the proposed algorithms. We implemented two variants of the algorithm, one optimized for the multicore architecture and the other optimized for disk-based computation. For a setting of ε, all implementations are guaranteed to produce a solution that is no worse than $(1+\varepsilon)H_n$ times the optimal solution, where $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ denotes the n-th Harmonic number.

5.1 Experimental Setup

Datasets. Our study uses a diverse collection of instances derived from various sources of popular graphs. Many of the datasets here are obtained from the datasets made publicly available by the Laboratory for Web Algorithmics at Università degli studi di Milano [8, 10]. These datasets are derived from directed graphs of various kinds in a natural way: each node v gives rise to a set and all nodes that v points to are members of the set corresponding to v. We assign unit weight to all the sets. Since the graphs are directed, some nodes may have nonzero out-degree but have in-degree 0. For the experiments, we consider such a node a set but not an element (it will never be covered). This asymmetry is the reason that the number of sets is not the same as the number of elements. We give a detailed description of each instance below and present a summary of these instances in Table 1.

—*livejournal-2008* is derived from user-user relationships on the LiveJournal blogging and virtual community site. Our dataset is a snapshot taken by Chierichetti et al. [11], where each set S is a user of the site and covers all users that are listed as S's friends.
— *webdocs* is a collection of web pages with directed links between them [18]. We derive a set system from this graph as described earlier.

Dataset	# of sets	# of elts.	# of edges	avg $\|S\|$	max $\|S\|$	Δ
webdocs	1,692,082	5,267,656	299,887,139	177.2	71,472	1,429,525
livejournal-2008	4,817,634	5,363,260	79,023,142	16.4	2,469	19,409
twitter-2010	40,103,281	41,652,230	1,468,365,182	36.6	2,997,469	770,155
twitter-2009	54,127,587	62,539,895	1,837,645,451	34.0	2,968,120	748,285
uk-union	121,503,286	133,633,040	5,507,679,822	45.3	22,429	6,010,077
altavista-2002-nd	532,261,574	1,413,511,386	4,226,882,364	7.9	2,064	299,007

Table 1. A summary of the datasets used in our experiments, showing for every dataset the number of sets, the number of elements, the number of edges, the average set size (avg $|S|$), the maximum set size (max $|S|$), and the maximum number of sets containing an element ($\Delta := \max |\{S \ni e\}|$).

— *twitter-2010* is derived from a snapshot taken in 2010 of the follower relationship graph on the popular Twitter network, where there is an edge from x to y if y "follows" x [22]. The set system is derived as described earlier.
— *twitter-2009* is an older, but larger, Twitter snapshot taken in 2009 by a different research group [20].
— *altavista-2002-nd* is the AltaVista web links dataset from 2002 provided by Yahoo! WebScope. The dataset has been preprocessed to remove dangling nodes, as suggested by experts familiar with this dataset[1].
— *uk-union* combines snapshots of webpages in the `.uk` domain taken over a 12-month period between June 2006 and May 2007 [9].

While coincidentally the real-world data sets we consider in this paper have about the same number of sets as the number of elements, our algorithms are not optimized to take advantage of this characteristic in any way.

5.2 Parallel Performance

The first set of experiments is concerned with the performance of our multicore-optimized program in comparison to existing sequential algorithms. These experiments are designed to test our implementation on the following important metrics:

1. **Solution's Quality.** The parallel algorithm should deliver solutions with no significant loss in quality when compared to the sequential counterpart;
2. **Parallel Overhead.** The parallel algorithm running on a single core should not take much longer than its sequential counterpart, showing empirically that it is work efficient; and
3. **Parallel Speedup.** The parallel algorithm should achieve good speedup[2], indicating that the algorithm can successfully take advantage of parallelism.

The baseline for the experiments is our own implementation of Cormode et al.'s disk-friendly greedy (DFG) algorithm [14]. DFG is a good baseline for this experiment because it achieves significant performance improvements over the standard greedy algorithm by making a geometric-scale bucketing approximation similar to ours. As previously shown, this approximation does not harm the solutions' quality in practice but makes it run much faster on both disk- and RAM- based environments. Our implementation of DFG closely follows the description in their paper but is further optimized for performance. Because of the fine tuning we made to the code, our implementation runs significantly faster than the numbers reported in Cormode et al. when all the data fits in RAM, taking in account the differences between machines. For this reason, we believe our DFG code is a reasonable baseline. We also implemented the standard greedy algorithm for comparison.

Evaluation Setup. Our parallel experiments were performed on a 40-core (with hyperthreading) Intel machine, consisting of *four* 2.4GHz 10-core E7-8870 Xeon processors, a 1066MHz bus, and 256GB of main memory. The machine is running Linux 3.2.0 (Red Hat). We compiled our programs with Intel Cilk++ build 8503 using the optimization flag `-O3`. The Cilk++ platform [23], in which the runtime system relies on a work-stealing scheduler, is known to impose only little overhead on both parallel and sequential code.

Results. Table 2 shows the performance of the three aforementioned RAM-based algorithms when run with $\varepsilon = 0.01$ (for DFG and parallel MaNIS). Several things are clear. *First, parallel* MaNIS *achieves essentially the same solutions' quality as both DFG and the baseline algorithm.* In fact, with ε set to 0.01 for both DFG and parallel MaNIS, all algorithms produce solutions of roughly the same quality—within about 1% of each other. We will note that, despite doing the most work, the standard greedy algorithm does not always yield the best solution. Our experience has been that the additional randomness that parallel MaNIS adds to the greedy algorithm often helps gain better solutions. In a number of datasets above, parallel MaNIS *does* yield the best-quality solutions.

Second, the parallel overhead in running MaNIS *is small.* This means that parallel MaNIS is likely to be faster than DFG even on a modest number of processors. As the numbers show, in all cases, parallel MaNIS is at most 1.8x slower than DFG when running on 1 core, and in 3 out of the 6 cases it runs in approximately the same time.

Third but perhaps most importantly, parallel MaNIS *shows substantial speedups on all but the small datasets.* The experiments show that MaNIS achieves upto 23.4x speedup with the speedup numbers ranging between 9x and 23.4x—except for the smallest dataset *webdocs* which obtains 6.9x speedup. This shows that the algorithm is able to effectively utilize available cores except when the datasets are too small to fully utilize parallelism (see discussion below).

To further understand the effects of the number of cores, we study the performance of parallel MaNIS on the *uk-union* dataset as the number of threads used is varied between 1 and 80 (all available threads). As Figure 3 shows, the running time performance of our algorithm scales well with the number of cores until at least 24 cores. After that, even though the performance continues to improve, the marginal

[1] See, e.g., `http://law.dsi.unimi.it/webdata/altavista-2002-nd/`
[2] This measures how much faster it is running on many cores than running sequentially.

Dataset	Standard Greedy		DFG		Parallel MaNIS		
	T_1 (sec)	# sets	T_1 (sec)	# sets	T_1 (sec)	T_{40h} (sec)	# sets
webdocs	24.2	406,399	6.46	406,340	6.66	0.96	406,343
livejournal-2008	9.80	1,120,594	4.53	1,120,543	5.58	0.62	1,120,599
twitter-2010	365	3,846,209	65.5	3,845,345	64.4	6.47	3,845,089
twitter-2009	689	5,518,039	97.3	5,516,959	87.6	7.63	5,517,864
uk-union	263	18,416,670	161	18,388,007	278	11.9	18,379,547
altavista-2002-nd	467	33,173,320	241	33,103,284	429	22.6	33,090,726

Table 2. Performance with $\varepsilon = 0.01$ of RAM-based algorithms: the standard greedy implementation, the disk-friendly greedy (DFG) algorithm of Cormode et al., and our MaNIS-based parallel implementation. We show the running time (**in seconds**) one core T_1 and on 40 cores with hyperthreading T_{40h} (80 threads), and the number of sets in the solutions.

benefit diminishes. Figure 4 demonstrates the break down of sequential running time into components and the speedup of these components on 40 cores. While most components in the *uk-union* dataset have near linear speedup, the bucket movement (i.e., filter) step, which takes 36.4% of the running time, is bandwidth constrained and achieves a speedup of only 17, limiting the overall speed up. Further experiments with filter and other microbenchmarks confirm this bottleneck.

For the smaller datasets, the parallel performance is additionally constrained by the speedup of the MaNIS and the `serialMaNIS` operations. The speedup of MaNIS and `serialMaNIS` depends on the size of the bucket on which they are executed. On large buckets of size exceeding 10^5, the speedup is limited by the bandwidth. On smaller buckets, the speedup is limited by the overhead of running multiple `cilk_for` operations. This overhead is increasingly prominent in the smaller datasets such as *webdocs*, in which case MaNIS achieves a speed up of only 5.5x compared to 25.6x on *uk-union*.

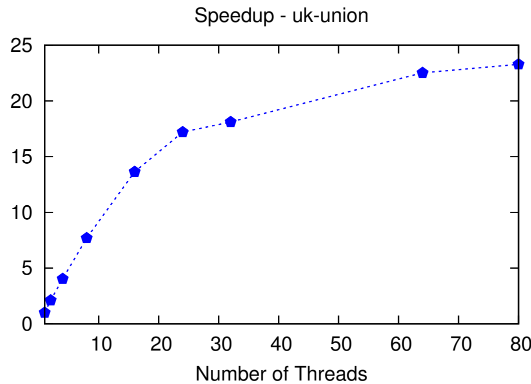


Figure 3. Speedup of the parallel MaNIS algorithm (i.e., how much faster is running the algorithm on n threads over running it sequentially) as the number of cores used is varied.

Max Cover. In the sequential setting, stopping the standard greedy set cover algorithm when it has found k sets gives the optimal $(1-1/e)$-approximation to max k-cover. An important feature of the parallel MaNIS algorithm is that it can be stopped early in the same way. To see how one might benefit from stopping the algorithm when the algorithm has found enough sets, we record the fraction of the total time when k sets are discovered. Presented in Figure 5 are plots from our 3 largest datasets (by the number of edges), *altavista-2002-nd*, *uk-union*, and *twitter-2009*. This experiment shows that although the rate varies between datasets, it is clear that most of the sets are added late in the algorithm; therefore, if the value of k of interest is small relative to the set cover solution's size, we can benefit from stopping early, which can often halve the running time. Chierichetti et. al. [12] present results for max k cover on map-reduce, however do not report any times so we were not able to compare.

5.3 Sequential Disk-based Performance

The second set of experiments deals with the performance of our disk-optimized implementation in comparison with existing disk-based algorithms. We are interested in evaluating the algorithms on the following metrics: (1) solutions' quality and (2) running time. Since the disk-optimized versions of both DFG and MaNIS implement the same algorithms as their parallel counterparts, their relative performance in terms of solutions' quality will be identical to the study conducted earlier for the parallel case. In the remainder of this section, we focus on investigating the running time as well as other performance characteristics of the disk-optimized MaNIS implementation.

Evaluation Setup. Our disk experiments were performed on a 4-core Intel machine although we only make use of a single core running at 2.66 Ghz. The machine is equipped with 8 GBytes of RAM and an Intel X25-M 160 GBytes SSD disk[3] (used both for input and as scratch space). There is a separate magnetic disk which we keep the OS Linux 2.6.38 (Ubuntu 11.04) and other system files. We compiled our programs with `g++` 4.5.2 using the optimization flag `-O3`.

We artificially limited the RAM size available to the set cover process to 512 *MBytes and carefully control all disk-access buffers to use only these* 512 *MBytes.* This may seem unrealistic at first, but this controlled setup models the types of machines available as embedded devices and computing nodes in low-power clusters (e.g., [1]) and provides a testbed for understanding the performance of these algorithms on such devices.

Table 3 reports the performance of the disk-based algorithms when run with $\varepsilon = 0.01$. On the larger graphs, the DFG algorithm did not complete within 40 hours. On the smaller sets, the disk based MaNIS is substantially faster (about 4x for *webdocs* and 39.6x for *livejournal-2008*). The

[3] Per Intel's specification, it has sustained sequential read and write bandwidths of 250 MB/s and 100 MB/s, resp.

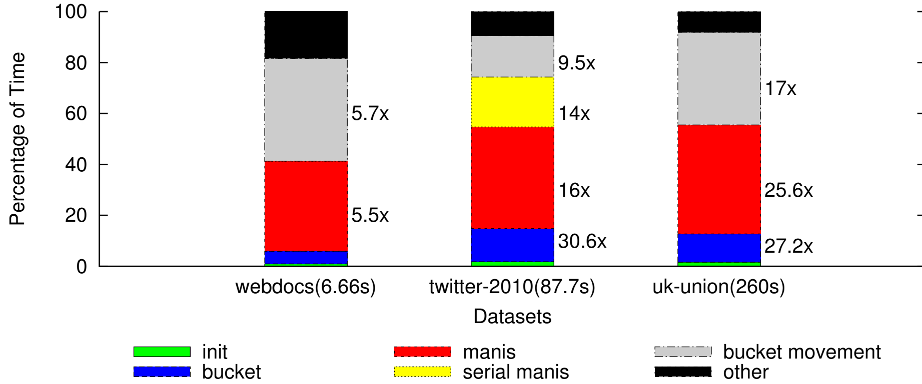


Figure 4. Timings and speedups of different components of the parallel set cover algorithms on `webdocs`, `twitter-2009`, `uk-union`. The sequential running time is shown next to the dataset's label, and the speedup of each major component is given next to the corresponding segment in the plot.

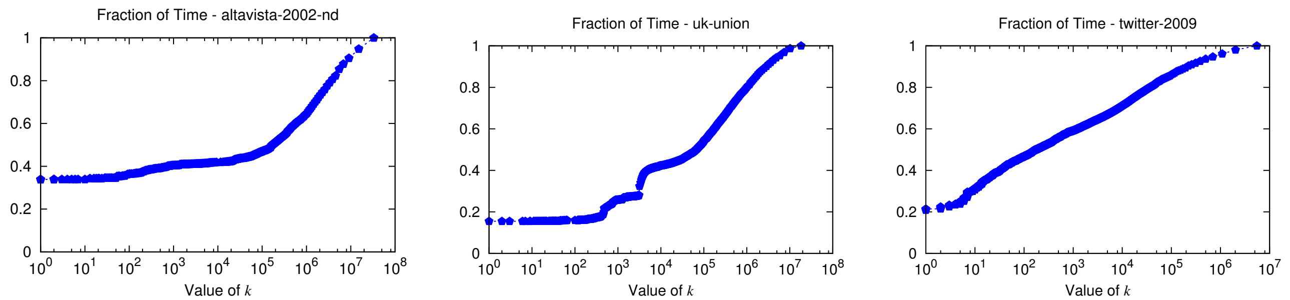


Figure 5. Max k-cover performance: The fraction of time spent to find the first k sets as the value of k is varied.

timing numbers may appear to be irregular, but a closer look at the results reveals patterns worthwhile mentioning:

When the bitmap representing the elements does not fully stay in fast memory (i.e., cache or RAM), the number of passes over the bitmap and the bitmap size crucially determine the performance of both algorithms. In DFG, which consults the bitmap every time it considers a set, this is lower-bounded by the number of sets in the output, whereas in disk MANIS, which batches "requests" to look up the bitmap and reduces them to one pass over it per MANIS round, this number is the total number of MANIS rounds summed across all buckets. This explains the difference in running-time patterns between the two algorithms on *webdocs* and *livejournal-2008*. For this reason also, the two Twitter datasets take roughly the same time to run disk MANIS despite significant differences in the number of sets in the solutions produced.

5.4 Effects of The Accuracy Parameter

In theory, the dependence on ε in the work bound is inversely proportional to $\log^3(1+\varepsilon)$, which, for small ε, is roughly $O(1/\varepsilon^3)$. This seems alarming because as we decrease ε (i.e., increase accuracy), $\frac{1}{\varepsilon^3}$ grows rather rapidly, rendering the algorithm unusable in no time; however, in practice, the situation is much better. As we decrease ε, we will observe more buckets but an even larger fraction of these buckets will be empty, reducing the efforts needed to run MANIS to process them and counteracting the increase in the number of buckets. Table 4 shows the effects of ε on *webdocs* for both the disk-optimized and parallel versions.

Algorithm	$\varepsilon = 0.1$	0.05	0.01	0.001
Disk MANIS	271	310	481	891
Parallel MANIS	0.36	0.47	0.96	2.06
# of sets	406,406	406,359	406,343	406,338

Table 4. Performance of MANIS-based algorithms on *webdocs* (**in seconds**) as ε is varied.

The numbers show that increasing the accuracy from $\varepsilon = 0.1$ to 0.001 (2 more digits of accuracy) increases the running time by less than 3x for the disk version and 6x for the parallel version. This trend seems to generalize across the datasets we have. More interesting, however, is the observation that ε only has small effect on the solutions' quality. Our experience on the datasets we have used is that we benefit more from spending computation time on different random priority orderings than adjusting ε to increase accuracy.

6. CONCLUSION

We presented a parallel cache-oblivious set-cover algorithm that has essentially the same approximation guarantees as the standard greedy algorithm. We implemented slight variants of the theoretical algorithm optimized for different hardware setups and provided experimental evaluation showing non-trivial speedups over existing algorithms while yielding the same solution's quality.

Dataset	DFG		Disk MaNIS	
	Time	# of sets	Time	# of sets
webdocs	32m50s	406,340	481s	406,367
livejournal-2008	3h5m	1,120,543	280s	1,120,599
twitter-2010	> 40 hrs	-	55m2s	3,845,089
twitter-2009	> 40 hrs	-	1h11m	5,517,864
uk-union	> 40 hrs	-	6h49m	18,379,547
altavista-2002-nd	> 40 hrs	-	13h27m	33,090,726

Table 3. Performance with $\varepsilon = 0.01$ of disk-based algorithms: the disk-friendly greedy (DFG) algorithm of Cormode et al., and our disk-based MaNIS implementation. We show the running time and the number of sets in the solutions.

Acknowledgments. This work is partially supported by the National Science Foundation under grant number CCF-1018188 and by generous gifts from IBM and Intel Labs Academic Research Office for Parallel Algorithms for Non-Numeric Computing. We thank the SPAA reviewers for their comments that helped improve this paper.

7. REFERENCES

[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[2] Bonnie Berger, John Rompel, and Peter W. Shor. Efficient *NC* algorithms for set cover with applications to learning and geometry. *J. Comput. Syst. Sci.*, 49(3):454–477, 1994.

[3] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPOPP*, pages 181–192, 2012.

[4] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, pages 355–366, 2011.

[5] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.

[6] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Near linear-work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. In *SPAA*, pages 13–22, 2011.

[7] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *SPAA*, pages 23–32, 2011.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[9] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[10] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[11] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009.

[12] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 231–240, New York, NY, USA, 2010. ACM.

[13] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):pp. 233–235, 1979.

[14] Graham Cormode, Howard J. Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *CIKM*, pages 479–488, 2010.

[15] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[16] Uriel Feige. A threshold of $\ln n$ for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[17] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[18] B. Goethals. Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data/.

[19] David S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[20] U. Kang, Brendan Meeder, and Christos Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *PAKDD (2)*, pages 13–25, 2011.

[21] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[23] Charles E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3), 2010. Springer.

[24] Sridhar Rajagopalan and Vijay V. Vazirani. Primal-dual *RNC* approximation algorithms for set cover and covering integer programs. *SIAM J. Comput.*, 28(2):525–540, 1998.

[25] Johannes Singler, Peter Sanders, and Felix Putze. The multi-core standard template library. In *Euro-Par*, pages 682–694, 2007.