An Efficient MapReduce Algorithm for Parallelizing Large-scale Graph Clustering

Inna Rytsareva, Ananth Kalyanaraman
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA, USA
Email: inna.rytsareva@email.wsu.edu, ananth@eecs.wsu.edu

***Abstract*—Identifying close-knit communities (or "clusters") in graphs is an advanced operation with a broad range of scientific applications. While theoretical formulations of this operation are either intractable or computationally prohibitive, practical algorithmic heuristics exist to efficiently tackle the problem. However, implementing these heuristics to work for large real world graphs still remains a significant challenge, owing to a combination of factors that include magnitude of the data, irregular data access patterns and compute-intensive operations to better the approximation. In this paper, we propose i) a novel MapReduce-based algorithm for a well known serial graph clustering heuristic called *Shingling*; and ii) a novel application of the method to cluster biological graphs built out of proteins and domains. Operating on an input graph that is simply represented as a list of edges, our algorithm uses a combination of shuffling and sorting operations, and pipelined MapReduce stages to implement the various phases of the algorithm. Preliminary results show linear scaling of the time-dominant phase up to 64 cores on a relatively small real world graph containing 8.41M vertices (8,407,839 proteins and 11,823 domains) and 11M edges (protein to domain connections). More importantly, MapReduce parallelization has allowed us to enhance the problem size reach by about two to three orders of magnitude (from 20K to 8M vertices) relative to our previous serial implementation, in roughly the same amount of time.**

***Index Terms*—Graph clustering; MapReduce algorithm; Shingling algorithm; dense subgraph detection; protein domain family.**

I. Introduction

Biological data, both naturally occurring and synthetically generated, lend themselves well to graph-based representations, where vertices can be used to represent the data points and edges (weighted or unweighted, directed or undirected) can be used to represent the relationship shared between data. Consequently, graph-based representations are a popular way to model problems in computational biology. Once modeled as a graph, various scientifically interesting questions can be posed on the data and they typically translate into performing some kind of graph operations — e.g., performing an Euler tour or Hamiltonian path for genome assembly, finding hubs and critical paths in gene regulatory networks, finding connected components to group expressed sequences (transcriptomics), and clustering, which forms the focal point for this paper.

Loosely defined, given an input graph $G(V, E)$ with n vertices and m edges, "*clustering*" is the act of grouping vertices into tight-knit communities, where each member of a group is closely linked to most (if not all) other members of the same group, and sparsely linked to members outside the group. This operation is sometimes also referred to as *community detection* but is different from graph partitioning, which involves partitioning the vertices into a pre-specified number of roughly equal-sized groups. In clustering, clusters are allowed to have different sizes, and the number of clusters and their size distribution are both unknown at input.

Clustering has a number of applications in computational biology. For instance: it can be used to reduce redundancy within sequence repositories; identify complexes within metabolic networks [2]; identify core groups of proteins that constitute a protein family [27], [31], [32] and in the process also help assign family memberships for newly found peptide candidates [32]; help in the construction of mass spectral libraries for peptides [18]; and can be used to condense the space of plausible computer-generated phylogenetic trees [24].

Despite its potential to address a broad range of problems, use of clustering in real world bioinformatics applications has been rather limited, with only a handful of projects benefiting from it at large-scale (e.g., [32]). The reason for this limited usage is the lack of scalable computational tools. Finding clusters is a data-intensive operation and it can easily become compute-intensive as well, depending on the heuristics used. The problem is equivalent to the problem of maximal, variable-sized dense subgraphs (or quasi-cliques), and theoretically speaking, several of the corresponding optimization problems are computationally hard problems [1], [10], [17] or with large degree polynomial methods [26], [27]. Therefore, faster approximation heuristics need to be used in practice. However, even such heuristics can be difficult to implement in parallel because of the irregular data access and computation patterns that they generate for different inputs.

In 2005, Gibson *et al.* developed an efficient graph clustering heuristic called *Shingling* [12]. Posed as a dense subgraph detection problem, their approach uses a randomized sampling method to iteratively identify and group vertices that share subsets of neighbors in common. In our earlier work, we implemented a serial version of this heuristic, and applied it in the context of metagenomic protein family detection [31]. Put briefly, this approach, called *pClust*, transforms the problem into one of bipartite graph clustering so that

the approach developed by Gibson *et al.* originally for web community detection can be used. The results [31], [30] on input sets of size up to 1.2 million amino acid sequences showed both run-time and quality (sensitivity) advantage over approaches that use other heuristics. Despite these advantages, the implementation of the clustering step (i.e., *pClust*) is serial and does not scale beyond a graph containing $15K$-$20K$ vertices on a desktop computer with 2 GB RAM due to memory requirement. To make it scalable for larger inputs, we had devised a two-step, albeit indirect, approach by which the large graph problem is first broken into connected components, and subsequently the sequential code is run on the individual connected components to output clusters. Owing to the simple observation that dense subgraphs cannot cut across multiple connected components and the expectation that the connected components in real world graphs tend to be large in number and small in sizes, this approach worked for clustering a set containing 1.2 million sequences (vertices). However, there is no guarantee it will work for larger inputs. In the worst case, the size of the largest connected component could become comparable to the size of the original input graphs.

In this paper, we present a new parallel algorithm for the *pClust* using the MapReduce paradigm [6]. We call this algorithm *"pClust-mr"*. The MapReduce model is suited for parallelizing this heuristic because of its capacity to handle large disk-resident data. However, one of the main challenges in the design is to contend with the irregular data access that arises while processing graph inputs under the distributed memory setting of MapReduce. The assumptions that the graph would fit in memory or can be evenly distributed/partitioned across the processor space are not practical. In *pGraph-mr*, we overcome these challenges.

More specifically, we make two contributions:

- We propose a novel MapReduce-based algorithm for the Shingling clustering heuristic. Operating on an input bipartite graph that is simply represented as a list of edges, our algorithm transforms the operations within the Shingling heuristic into a combination of standard MapReduce primitives such as *map*, *reduce* and *group/sort*. The algorithm is a pipelined MapReduce implementation. The novelty of the design comes from the following fact: Even though the parallel algorithm solves the same problem and maintains the overall structure of the serial method, the individual stages are implemented in a significantly different manner in order to efficiently extract parallelism under the MapReduce framework.
- We introduce a new formulation for clustering protein sequences based on domains[1], and present a novel application of *pClust-mr* to address this formulation. Preliminary results show *pClust-mr* scales linearly in its most time-consuming phase on up to 64 cores on a small real world graph containing 8.41M vertices (8,407,839 proteins and 11,823 domains) and 11M edges (protein to domain connections). More importantly, the parallel method has allowed us to enhance the problem size reach by about two to three orders of magnitude (20K to 8M vertices), and reduce the time to solution from days to minutes — for instance, *pClust-mr* solves the 8.41M vertices input problem in ∼20 minutes, for which the serial implementation [31] is expected to take ∼70 hours if sufficient memory is made available.

[1]A "domain" is a contiguous stretch of a protein sequence that is structurally (and often times, also functionally) conserved among multiple protein sequences.

The paper is organized as follows: Section II presents a brief overview of the related literature on clustering and dense subgraph detection. In Section III, we first describe the sequential clustering algorithm and then present our MapReduce algorithm and its analysis. Experimental results are presented in Section IV, and Section V concludes the paper.

II. Background and Related Work

The problem of finding a densest subgraph within an input graph is solvable in polynomial time [5], [13], [20]. However, the more practically appealing constrained variants of this problem, viz. of finding a densest subgraph of size equal to k, or at least k, or at most k, have all shown to be NP-Hard [1], [10], [17]. Our problem represents a more generalized version of these variants, wherein the goal is to find multiple, variable-sized maximal dense subgraphs, satisfying density and size cutoffs. Consequently, approximation heuristics need to be pursued. Dense subgraph detection problems can also be defined over bipartite graphs. This way of modeling the problem is particularly effective when relationships are defined over data of two different types, and find frequent usage in the context of web communities in internet data (e.g., [12]). It turns out that the bipartite graph formulations are also NP-Hard [10], [21].

There is a rich body of clustering related literature in the context of biological applications. For a considerable number of applications simple agglomerative techniques (e.g., neighbor joining) or single linkage clustering suffices in practice (reviewed in [7], [8], [15]). A different set of applications benefit from graph based clustering formulations that aim to detect tight communities within biological data (e.g., [2], [9], [14], [22], [32]).

Independently, in other areas of computing such as social and cyber networks, numerous algorithms have been developed for community detection (reviewed in [19], [21], [25]). M.E.J. Newman, in his pioneering work on discovering community structure from networks [26], developed a divisive clustering method that detects and removes edges, one at a time, that are most likely to cut across cluster partitions. To detect such edges, the approach calculates the betweenness centrality index for all edges in the graph. However, removal of an edge introduces the need to recompute the centrality index for all edges. While this approach has been demonstrated to be highly effective in discovering community structure [26], the cost of computing centrality index and the need for recomputing after every step make the algorithm slow ($\Omega(n^3)$ even for sparse graphs with n vertices) and practical

for only up to $n \approx 10^4$ on single compute nodes. Nevertheless, there are shared memory parallel algorithms such as [23] for efficiently calculating betweenness centrality on graphs. A different approach [27] works on weighted graphs, where edge weights are distance measures, and uses Minimum Spanning Trees (MST) for clustering by taking advantage of the property that closely related groups tend to map to subtrees within an MST. However, this method can also be time consuming ($\Omega(n^2)$) and the method has not been compared with other methods making it difficult to assess its quality.

Gibson *et al.* developed the Shingling approach for identifying web communities [12]. The underlying method (described in Section III-C) uses a random sampling procedure and secondary sorting to determine dense subgraphs. In [31], we adapted this method to work for graphs constructed using protein sequences as vertices and the presence or absence of pairwise full-length similarity (or homology) to mark the presence or absence of edges, respectively. The method showed improved sensitivity [31], [30] when compared to other methods [32]. Yet the scalability of the serial implementation was bounded by the size of the graph that can fit in the local memory (∼20K based on our experience). And the running time is dominated by a sorting step that sorts $O(n \times c)$ values, where n is the number of vertices and c is a parameter (typically ≥ 100), and this could mean several hours to even days for larger values of n ($> 10^6$). For instance, on an input with 20K vertices, the serial code took 10 minutes. Even assuming a perfect linear scaling with input size, this implies ∼70 CPU hours on the input with 8.41M vertices. The MapReduce-based algorithm presented in this paper reduces this run-time from days to minutes, and with a disk storage requirement of $O(m + n)$, where m is the number of edges, the method enhances the problem size reach by a couple of orders of magnitudes than was possible before.

III. Methods

A. Problem definition

Let $G = (V_l, V_r, E)$ denote an undirected bipartite graph with n vertices ($n = |V_l| + |V_r|$) and m edges. Let $\Gamma(u) = \{v \mid (u, v) \in E\}$ be the set of *out-links* of a vertex u. For the purpose of convention, we will use u to denote a vertex from V_l and v to denote a vertex from V_r. The maximality clause and the undirected nature of edges ensure that property also holds from V_r' to V_l'. Let a *shingle* [4] be a constant-size subset of vertices from V_l or V_r.

Defined loosely, a *dense subgraph* in a bipartite graph is a maximal subgraph containing subsets $V_l' \subseteq V_l$ and $V_r' \subseteq V_r$ such that each vertex in V_l' is connected to most of the vertices in V_r' and sparsely (if at all) connected to vertices in $V_r \setminus V_r'$. In other words, the vertices in V_l' share most of their out-links in V_r'.

Given $G(V_l, V_r, E)$, our goal is to find a set of maximal, variable-sized dense subgraphs within G. As part of this definition, note that we do *not* enforce that the output dense subgraphs be disjoint. In other words, the output dense subgraphs, or simply "clusters" as we will refer to them henceforth, need not represent a partition of the input graph.

B. Domain-based protein clustering: A motivating biological application

In the realm of protein sequences, proteins are said to contain *domains*, where each *domain* is a contiguous stretch that is conserved among multiple protein sequences. One can think of a domain as a substring that is shared (with a few variations) among multiple protein sequences. It is understood that each domain represents an autonomous functional unit and therefore, proteins sharing a given domain are likely to be linked functionally. Consequently, there are databases, the most notable one being Pfam [11], [28], which catalog proteins and their domains. Currently, the practice is to group the set of proteins that share a single domain into a protein "family" [28]. However, proteins could share multiple (variable number of) domains in common, implying a variable degree of functional correlation, and current databases do not provide a familial classification that takes into account this variability.

Following the problem formulation in Section III-A, we can construct an input bipartite graph $G = (V_l, V_r, E)$ can be built such that V_l is the set of all domains and V_r is the set of all protein sequences, and $(u, v) \in E$ if domain u is contained in protein v. Subsequently, a clustering in G would represent a grouping of protein sequences based on multiple shared domains. Such an output could help better classify the protein space based on domain knowledge and enhance functional characterization.

Treating the above problem as a motivating application, we address the problem of bipartite dense subgraph detection for large graphs. In what follows, we first describe the Shingling heuristic [12] for bipartite graph clustering and its serial implementation, *pClust* [31]. Subsequently, we present our new MapReduce based parallel algorithm for *pClust*.

C. The serial algorithm

A brute-force way to detect vertices that form the core of a dense subgraph would be to compute $\frac{|\Gamma(u_i)| \cap |\Gamma(u_j)|}{|\Gamma(u_i)| \cup |\Gamma(u_j)|}$ for every pair of vertices $u_i, u_j \in V_l$. The Shingling heuristic takes a randomized sampling approach to reduce this search space. More specifically, it obtains random samples of size s from $\Gamma(u)$ for every vertex $u \in V_l$, and compares them against one another. If two vertices are part of the same dense subgraph, then by definition they should also share most of their out-links and hence with a high probability also expected to share shingles [3]. However, this cannot be guaranteed because the shingles are small, fixed-size samples obtained randomly (using a pair of random numbers A and B). Therefore, to improve the probability that such vertex pairs are detected, the algorithm generates shingles over c random trials.

Generation of c shingles for any vertex $u \in V_l$ that has at least s out-links is achieved as follows: First, c random permutations of the vertices in $\Gamma(u)$ are obtained using c pairs of random numbers $\{<A_j, B_j> | j \in [1, c]\}$. The top s

elements within each permutation are then said to represent a shingle.

The above heuristic is implemented in *pClust* in three phases:

Shingling Phase I: Using input $G(V_l, V_r, E)$ in its adjacency list form, the algorithm first generates c shingles for each vertex in V_l as described above. Let s_j denote a shingle generated for some vertex during the jth random trial, and assume that it is in an integer representation obtained using a hash function. Since the same shingle s_j could have been generated by multiple vertices in V_l, a sorting is done to gather all vertices that generated each shingle. Let $L(s_j)$ denote the set of vertices which generated a shingle s_j. The algorithm then outputs tuples of the form $<s_j, L(s_j)>$. Note that these tuples collectively define a new bipartite graph $G_I(S_1, V_l', E')$ in its adjacency list form, such that S_1 represents the set of distinct shingles generated during this phase, and $V_l' \subseteq V_l$ represents the subset of vertices that contributed to at least one shingle. Therefore, the output of this phase is G_I. We call the shingles in S_1 *first level shingles*.

Shingling Phase II: Using G_I as the new input, the algorithm executes the same series as steps as in Phase I. This generates a new bipartite graph $G_{II}(S_2, S_1', E'')$ in its adjacency list form, such that S_2 represents the new set of shingles generated during this phase (referred to as the *second level shingles*), and $S_1' \subseteq S_1$ represents the subset of first level shingles that contributed to at least one second level shingle in S_2.

Phase III - Connected component detection: In the final reporting step, all connected components in G_{II} are reported. Note that the connected components will be defined by first to second level shingle connections. To enumerate connected components, this step uses the classic union-find data structure. Consequently, the union of vertices in G within each connected component of G_{II} is reported as the output set of dense subgraphs.

The implementation has the following runtime complexity by stages: i) $O(n \times c \times s^2 + T_{sort}(n \times c)))$ for the Shingling Phase I, where T_{sort} denotes the sorting time. Since *pClust* uses quicksort, expected runtime is $T_{sort}(n \times c) = O(n \times c \times \log(n \times c))$; ii) for Shingling Phase II, the runtime complexity is $O(|S_1| \times c \times s^2 + T_{sort}(|S_1| \times c)))$, where $c \leq |S_1| \leq n \times c$ depending on the input; iii) for Phase III, the runtime is $O((|S_1| + |S_2|) \times s \times \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function which is a small constant for all practical purposes.

The peak memory complexity of the algorithm is $O(max\{m+n, |S_1|, |S_2|\})$, and is $\Theta(n \times c^2)$ in the worst-case.

D. pClust-mr: A MapReduce algorithm for pClust

In designing our MapReduce algorithm for the above dense subgraph detection algorithm, we preserved the overall algorithmic structure in *pClust* but implemented each phase in a way that is better suited to the distributed memory setting of MapReduce. In what follows, we present the proposed parallel algorithms along with associated design challenges for the three phases.

1) Shingling Phases I and II: The serial algorithm assumes an adjacency list format. This provides direct and easy access to all of the out-links for any given vertex to generate the shingles. However, an adjacency list representation may not be appropriate under a MapReduce for a couple of reasons. Assuming each vertex's list occupies a line in the input file, the MapReduce framework would distribute the input lines in parallel to the map tasks. Because a line is never split under MapReduce, this requires that the memory for storing at least one line of the input should be available on the compute node running a map task. In the worst-case, a vertex's adjacency list could be $\Theta(n)$. For large values of n this memory requirement may not be a scalable option. Secondly, the adjacency lists of vertices could vary in length, implying a potential scenario where map tasks with long lists could become a parallel run-time bottleneck. An alternative option is to use the adjacency matrix representation, which would at least make the line lengths uniform, but it is too expensive in terms of storage ($\Theta(n^2)$).

Another and a justifiably better option is to store the graph as a *simple list of edges*. This representation is naturally suited for MapReduce because each edge $<u, v>$ is in the form of a <key,value> tuple that conforms to the input/output types of the framework. This representation also automatically keeps the line length short and uniformly sized. However, the challenge rests on being able to generate a shingle because the edges of a given vertex could potentially be scattered across different map tasks.

We devised an algorithm that uses the edge list representation and overcomes all the above outlined challenges. Our algorithm is illustrated in Figure 1. The main idea is as follows: The input is a simple list of edge tuples $<u, v>$, where $u \in V_l$ and $v \in V_r$, one edge per line. Instead of attempting to generate the c shingles corresponding to each input vertex u at the Map phase, we defer that task to the Reduce phase and instead only generate all the c forms for an out-link v, viz. $\{v^1, v^2, \ldots v^c\}$ at the Map phase. Basically, each mapper at any given point of time, takes as input an edge tuple $<u, v>$ and emits c tuples of the form $<u, v^j>$, where $j \in [1 : c]$. This can be implemented as a strictly local operation, assuming all the c random number pairs ($<A_j, B_j>$) are made available at initialization time at each mapper.

Next, the tuples emitted by the mappers are grouped using the source vertex u as the intermediate key. This sends the list of all tuples generated for a given u to a single reducer. Note that the length of this list could be anywhere in the range $[c \ldots (c \times |\Gamma(u)|)]$.

At the reducer designated for u, if one were to store the entire list of its tuples to sort and generate the c shingles, it would imply a memory complexity of $\Theta(c \times n)$ in the worst-case. Therefore, we devised a different approach in which the

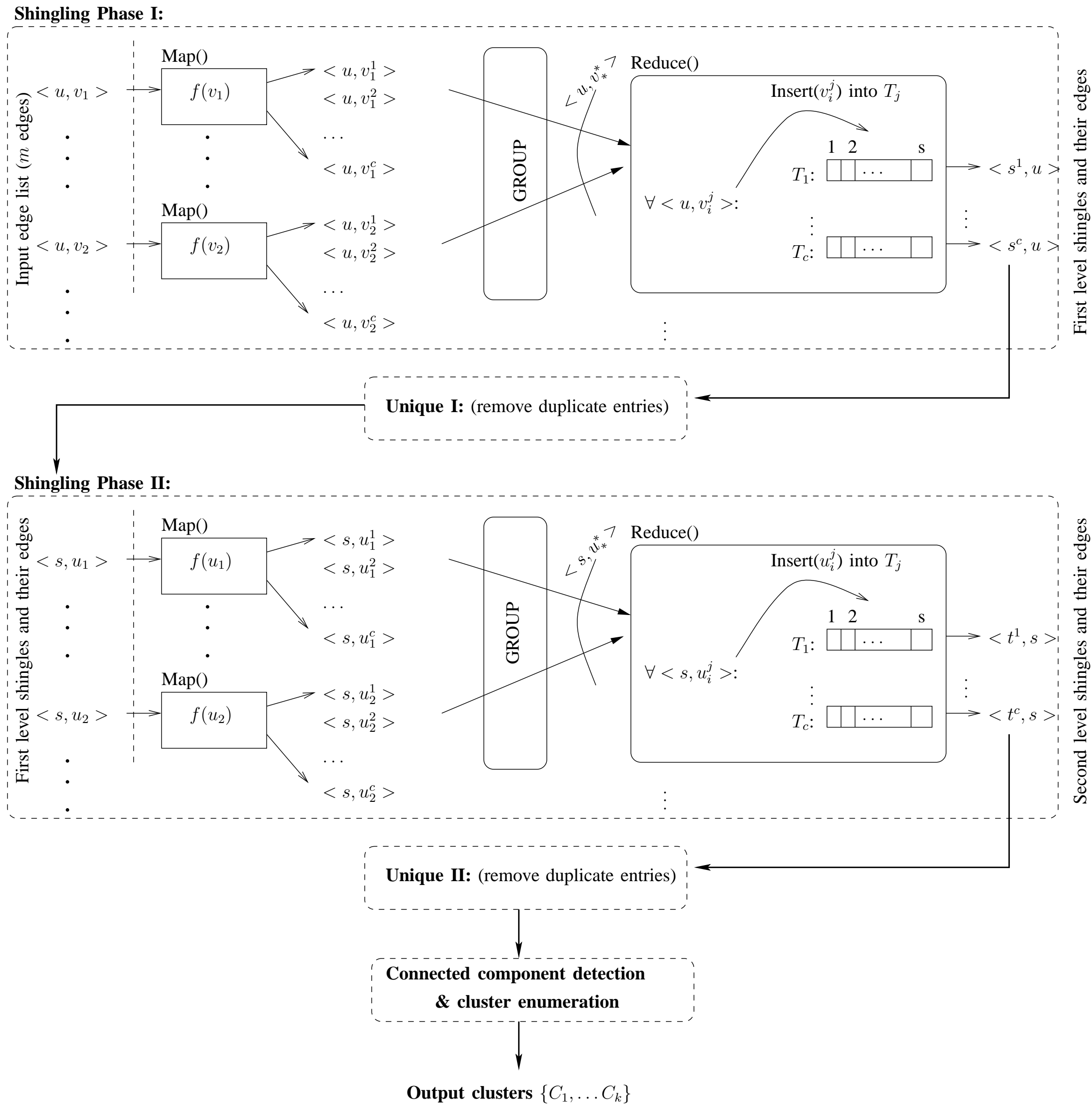


Fig. 1. Illustration of the *pClust-mr* algorithm. The figure shows the detailed MapReduce algorithm for Shingling Phases I and II. The function $f()$ (inside Map()) takes as argument a vertex v in the form of its integer id and generates c forms of v, viz. $\{v^1, v^2, \dots v^c\}$; where $v^j = (A_j \times v + B_j)\%P$, $j \in [1 \dots c]$ corresponds to the jth trial number, $< A_j, B_j >$ denote the jth pair of random numbers, and P is a big prime number. Unless otherwise explicitly noted, the tuples emitted by the mappers are grouped by the first field — e.g., for $< u, v >$, the intermediate key is $< u >$.

memory complexity per reducer becomes independent of the length of the tuple list. This is achieved as follows: Every reducer maintains a set of c arrays of size s each. We call these "trial arrays", and denote the arrays as $T_1 \dots T_c$. Each T_j keeps track of the minimum s elements seen so far for that trial. When a new $<u, v^j>$ tuple is streamed in, the element v^j is *inserted* into T_j so as to maintain the sorted order within T_j. Given the small values expected for s (typically, $s \leq 5$ in practice), this is implemented using a simple linear scan as in insertion sort, taking at most s comparisons per insert. After all the tuples have been processed, the elements stored in the trial arrays represent the c target shingles. The reducer then assigns each shingle an integer id (using a hash function). To ensure that different permutations of the same s elements are assigned the same shingle id, we internally sort each shingle by its constituent vertex names. This adds another $O(s)$ to the processing time per trial at output. Consequently, the reducer emits tuples of the form $<s, u>$ from each T_j, where s is a first level shingle and the u is a vertex that generated it.

Note that the global set of all tuples $<s, u>$ emitted by all reducers is the edge list representation for graph G_I. It is possible that this list emitted by the reducers contain some duplicate tuples (as different trials of the same source vertex could have identified the same shingle). To eliminate such duplicates, we implemented a simple Map-Reduce phase where identical tuples (lines) are grouped and only one copy of each tuple is emitted. This phase is identified as the *Unique I* phase in Figure 1. The resulting non-redundant list of edges

is passed as input to Phase II. It should be easy to observe that the algorithm for Phase II is identical to that of Phase I.

2) Analysis of the Shingling phases: The disk storage complexity is $\Theta(m)$, because the algorithm operates using a simple edge list. The memory complexity at each mapper is $O(1)$ and each reducer is $\Theta(c \times s)$. As for the run-time complexity, let p denote the total number of processors. Let us also assume, for sake of analysis, that the number of map tasks (p_m) plus the number of reduce tasks (p_r) is equal to p. Then the run-time complexity for each mapper is expected to be $O(\frac{m \times c}{p_m})$. As for the reducers, assuming a balanced distribution of the grouped tuples, each reducer is expected to take $O(\frac{m \times c \times s}{p_r})$ time. The overall run-time complexity for Phase I is expected to be $O(\frac{m \times c}{p_m} + T_{group}(m \times c) + \frac{m \times c \times s}{p_r})$, where $T_{group}(m \times c)$ represents the time for the MapReduce framework to group $m \times c$ tuples. In practice, reducers can run concurrently with mappers because for grouping they need not wait for mappers to finish. Therefore, the time to group and mapper run-times may actually dominate.

Phase II's analysis is similar except that the input number of edges in G_I is bounded by $O(n \times c)$ (worst-case represented by a sparse graph). Therefore, a run-time complexity of $O(\frac{n \times c^2}{p_m} + T_{group}(n \times c^2) + \frac{n \times c^2 \times s}{p_r})$ follows. In practice, assuming the graph is well connected, this run-time can be expected to be significantly smaller than the theoretical bound.

E. Phase III: Connected component detection

For connected component detection, the idea used of using an irregular data structure such as the union-find is not suitable under distributed memory setting. For this paper, we developed a MapReduce algorithm which implements a technique that was originally used for conducting Breadth First Traversals for the MPI model [29]. The MapReduce algorithm is illustrated in detail in Figure 2. Due to space constraints, we describe only the main idea behind the technique. Define the *label* of vertex u to be equal to the maximum vertex id in u's connected component. Since the connected components are unknown initially, the algorithm starts by assigning the labels at every vertex to itself. Then an iterative approach follows: At the end of iteration $\#0$, each vertex exchanges information with its immediate neighbors and accordingly updates its label. Subsequently, the labels from iteration $i-1$ are used to update the labels at iteration $\#i$. The algorithm terminates when it the set of labels converges.

We note here that, after we developed this approach under MapReduce, we found out that a nearly-identical algorithm has already been developed by Kang *et al.* [16] The only difference between our algorithm and their's is that our algorithm has a better memory complexity at the reducers, but uses a MapReduce sort (as opposed to a MapReduce group). More specifically, our reducer in $UpdateLabels(i)$ will work even under a streaming model, implying a $O(1)$ memory requirement (as opposed to $O(n)$ in [16]).

Input label	Number of edges	Number of vertices		
		# proteins	# domains	Total
$11M$	11,4167,76	8,407,839	11,823	8,419,662
$8M$	8,388,608	6,639,087	3,537	6,642,624
$4M$	4,194,304	3,681,067	665	3,681,732

TABLE I
INPUT DATA STATISTICS.

IV. Experimental Results

A. Experimental setup

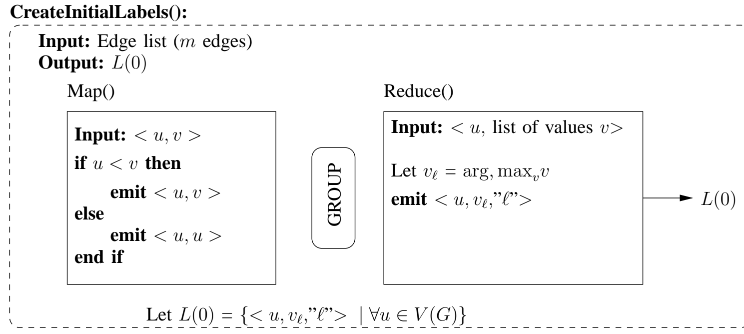
We used the *Magellan* Hadoop cluster at National Energy Research Scientific Computing Center (NERSC) as our experimental platform. The cluster has 78 nodes with a total of 624 cores dedicated for Hadoop, where each node has 2 quad cores Intel Nehalem 2.67 GHz processors and 24 GB DDR3 1333 MHz RAM. These nodes run Cloudera's distribution for Hadoop 0.20.2+228.

As for the input, we downloaded the *Pfam-A* database from the Pfam website [28]. The largest input graph constructed out of this database contains 8,419,662 vertices (=11,823 domains + 8,407,839 proteins), and 11,416,776 edges connecting them. Smaller subgraphs were extracted from this large graph for scalability studies. Table I shows the input statistics. A value of $s = 2$ and $c = 100$ was used in all our experiments, based on our earlier experiments with *pClust* [31], [30].

B. Performance results

For the purpose of this paper, we primarily analyzed the performance of Shingling Phase I in *pClust-mr*. This is because we expect Shingling Phase I to be the time-dominant phase (as will be later shown in Table III). Table II shows Shingling Phase I's run-time as a function of the number of processors used (by varying the number of map tasks from 16 to 128). The results show that our implementation scales linearly up to 32 cores for the smaller inputs 4M and 8M. When the input size was increased to 11M, the linear scaling behavior extends to 64 cores. The deterioration in scaling for larger number of cores is expected because for those processor sizes the problem size becomes too small to benefit from parallelism, and the parallel overheads of the MapReduce framework start dominating. Figure 3, which shows the relative speedup calculated using the 16 tasks run as the reference, confirms this scaling trend. A peak speedup of $\sim$90x is obtained for the 11M input running using 128 map tasks. Figure 4 shows the parallel efficiency.

MapReduce framework supports a feature whereby one could run an arbitrary number of map and reduce tasks, independent of the number of cores. This can be used to determine task granularity that is empirically optimal. To study this effect, we varied the number of number map tasks from 64 up to 16K. The results are shown in Figure 5. Note that in our experimental Hadoop cluster, there is a system-imposed cap of 480 cores that can be used for map tasks. It can be observed that the run-time decreases gradually until 2K map tasks and then starts to increase again. The reason for the



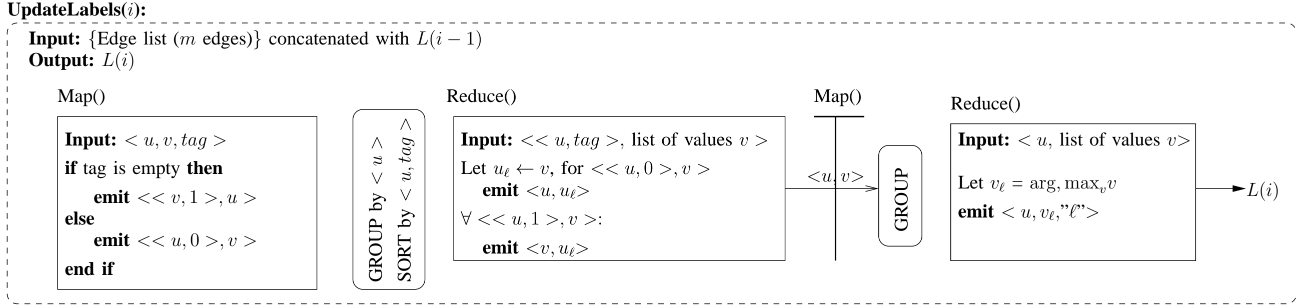


Fig. 2. Illustration of the MapReduce algorithm for connected component detection. The algorithm terminates when $L(i) == L(i-1)$.

Number of map tasks and reduce tasks	Time		
	4M	8M	11M
$(p_m = 16, p_r = 11)$	2,188	4,397	6,874
$(p_m = 32, p_r = 11)$	985	1962	3,599
$(p_m = 64, p_r = 11)$	684	1,223	1,701
$(p_m = 128, p_r = 37)$	531	1,740	1,241

TABLE II
THE RUN-TIME (IN SECONDS) OF SHINGLING PHASE I ON VARIOUS INPUTS AND SYSTEM SIZES. ALL RUNS WERE PERFORMED USING $s = 2$ AND $c = 100$.

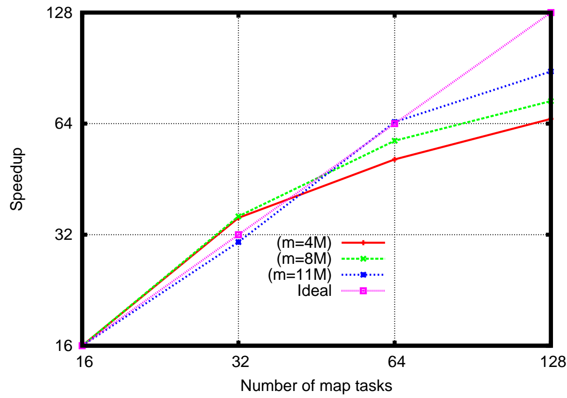


Fig. 3. Speedup of the Shingling Phase I up to 128 map tasks.

initial decline in run-time is because with increased number of map tasks, reducers are able to start earlier. However that benefit is soon offset by the increase in system overhead that is required to manage an increasing number of map tasks.

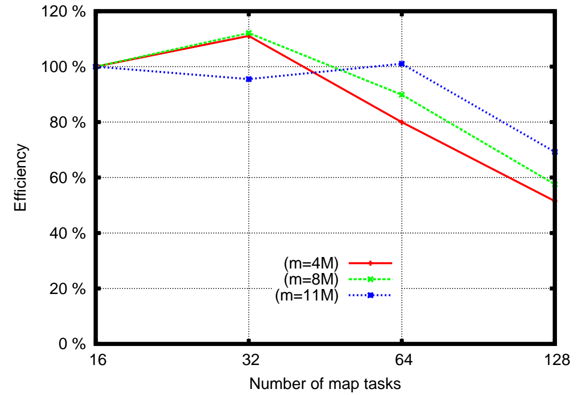


Fig. 4. Parallel efficiency of Shingling Phase I.

For this set, we find that the empirical optimal value for task granularity is occurring approximately at 5,500 edges per map task ($\approx \frac{11M}{2K}$).

Table III shows the phase-wise breakdown of the total runtime for the 11M input as a function of the number of map tasks. It can be clearly seen that Shingling Phase I is the dominant phase and that it scales linearly up to 64 map tasks. The reason for the lack of scaling of the remaining phases is primarily because the corresponding input sizes were too small to gain from parallelism. In the immediate future, we plan to conduct more experiments with much larger scale inputs and study the performance of the other phases as well.

V. CONCLUSIONS

In this paper, we presented a novel MapReduce algorithm called *pClust-mr* that can be used to identify dense subgraphs

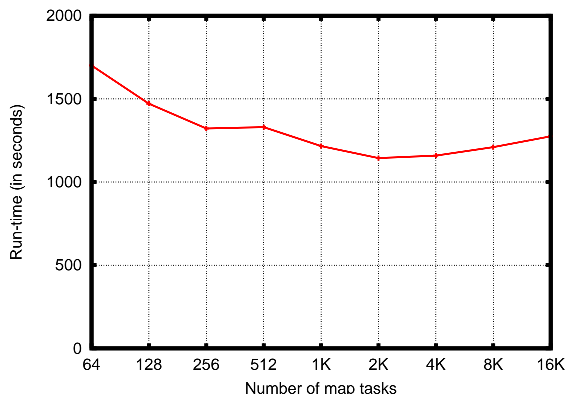


Fig. 5. Effect of varying the number of map tasks on Shingling Phase I ($m = 11M$). It can be observed that the optimal setting for the number of map tasks is 2K for this input.

Phase	Number of map tasks			
	16	32	64	128
Shingling Phase I	6,874	3,599	1,701	1,241
Unique I	26	34	25	26
Shingling Phase II	982	797	682	319
Unique II	26	28	24	26
Connected component detection	679	733	705	763
Total	8,587	5,191	3,137	2,375

TABLE III
BREAKDOWN OF RUNTIME (IN SECONDS) BY THE DIFFERENT PHASES OF THE ALGORITHM FOR THE INPUT GRAPH ($m = 11M$). THE NUMBER OF REDUCERS WAS KEPT FIXED AT 11 FOR ALL RUNS EXCEPT FOR RUNS WITH 128 MAP TASKS, THE NUMBER OF REDUCERS WAS INCREASED TO 37.

from bipartite graphs. We also presented a novel application of this method to a problem pertaining to clustering protein sequences based on domain knowledge. Although the research is at an early stage, preliminary results presented in this paper indicate linear scaling behavior of the most dominant phase (Shingling Phase I). More importantly, the results demonstrate the effectiveness of the MapReduce framework for solving this irregular graph problem, and the potential of the proposed method to scale to much larger input sizes. Several studies and extensions have been planned as part of future work. Importantly, we plan to test out our implementation for larger real world graphs, optimize the performance of the remaining phases, and analyze the qualitative and scientific merits of protein-domain clustering.

ACKNOWLEDGMENT

This research was supported in parts by DOE award DE-SC-0006516 and NSF grant IIS 0916463.

REFERENCES

[1] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. *Lecture Notes in Computer Science*, 5427:25–36, 2009.
[2] G. Bader and C. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4:2, 2003.
[3] A.Z. Broder, M. Charikar, A. Frieze and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:630–659, 2000.
[4] A.Z. Broder, S. Glassman, M. Manasse and G. Zweig. Syntactic clustering of the web. *WWW6/Computer Networks*, 60(8-13):630–659, 2000.
[5] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. *Lecture Notes in Computer Science*, 1913:1157–1166, 1997.
[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113, 2008.
[7] P. D'haeseleer. How does gene expression clustering work? *Nat Biotech*, 23:1499–1501, 2005.
[8] S. Emrich, A. Kalyanaraman, and S. Aluru. Chapter 13: Algorithms for large-scale clustering and assembly of biological sequence data. *Handbook of computational molecular biology*, CRC Press, 2005.
[9] A.J. Enright, S. Van Dongen, and S.A. Ouzounis. An efficient algorithm for large-Scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.
[10] U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. *Algorithmica*, 29:410–421, 2001.
[11] R. Finn, J. Mistry, J. Tate, *et al.* The Pfam protein families database. *Nucleic Acids Research*, 38(1):D211–D222, 2010.
[12] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. International Conference on Very Large Data Bases*, pp. 721–732, 2005.
[13] A.V. Goldberg. Finding a maximum density subgraph. *Technical Report CSD-84-171*, UC Berkeley, 1984.
[14] H. Jeong, S.P. Mason, A. Barabasi, and Z.N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41–42, 2001.
[15] A. Kalyanaraman and S. Aluru. Chapter 12: Expressed Sequence Tags: Clustering and applications. *Handbook of computational molecular biology*, CRC Press, 2005.
[16] U. Kang, C. E. Tsourakakis and C. Faloutsos. PEGASUS: A peta-scale graph mining system – implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM 2009)*, pp. 229–238, 2009.
[17] S. Khuller, B. Saha, S. Albers, *et al.* On Finding Dense Subgraphs. *Automata, Languages and Programming*, Springer Berlin / Heidelberg, pp. 597–608, 2009.
[18] H. Lam, E.W. Deutsch, J.S. Eddes, *et al.* Building consensus spectral libraries for peptide identification in proteomics. *Nat Meth*, 5:873–875, 2008.
[19] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Phys. Rev. E*, 80:056117, 2009.
[20] E. Lawler. Combinatorial optimization - networks and matroids. *New York: Holt, Rinehart and Winston*, 1976.
[21] V.E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A Survey of Algorithms for Dense Subgraph Discovery. *Managing and Mining Graph Data*, A.K. Elmagarmid, C.C. Aggarwal, and H. Wang, Eds., Springer US, pp. 303–336, 2010.
[22] H. Ma and A. Zeng. The connectivity structure, giant strong component and centrality of metabolic networks. *Bioinformatics*, 19:1423–1430, 2003.
[23] K. Madduri, D. Ediger, K. Jiang, *et al.* A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. *Proc. Workshop on Multithreaded Architectures and Applications*, Rome, Italy, May 29, 2009.
[24] S. Matthews and T. Williams. MrsRF: an efficient MapReduce algorithm for analyzing large collections of evolutionary trees. *BMC Bioinformatics*, 11:S15, 2010.
[25] M.E.J. Newman. Networks: An introduction (Chapter 11). *Oxford University Press*, 2010.
[26] M.E.J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.
[27] V. Olman, F. Mao, H. Wu, and Y. Xu. A parallel clustering algorithm for very large data sets. *IEEE/ACM Transaction on Computational Biology and Bioinformatics*, 5(2):344–352, 2007.
[28] Pfam - Protein family database. http://pfam.sanger.ac.uk/. Last date accessed (10/3/2011).
[29] A. Yoo, E. Chow, K. Henderson, *et al.* A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proc. ACM/IEEE Supercomputing*, pp. 25–35, 2005.

[30] C. Wu. Parallel algorithms for large-scale computational metagenomics. *Ph.D. dissertation*, Washington State University, Pullman, WA, USA, Spring 2011.

[31] C. Wu, and A. Kalyanaraman. An efficient parallel approach for identifying protein families in large-scale metagenomic data sets. In *Proc. ACM/IEEE conference on Supercomputing*, pp. 1–10, 2008.

[32] S. Yooseph, G. Sutton, D.B. Rusch, *et al.* The Sorcerer II Global Ocean Sampling expedition: expanding the universe of protein families. *PLoS Biology*, 5(3):432-466, 2007.