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 (protein to domain connections). While theoretical formulations of this operation 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, such heuristics can be difficult to implement in parallel because of the irregular data access and computation patterns that they generate for different inputs.

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, 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 groupsort. 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.

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

\(^1\)A “domain” is a contiguous stretch of a protein sequence that is structurally (and often times, also functionally) conserved among multiple protein sequences.
A brute-force way to detect vertices that form the core of a dense subgraph would be to compute \( \frac{\Gamma(u)_s^2}{\Gamma(u)} \) for every pair of vertices \( u, u_j \in V_l \). 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 \( s \) 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.
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 \( G(V_t, V_r, E) \) in its adjacency list form, the algorithm first generates \( c \) shingles for each vertex in \( V_t \) as described above. Let \( s_j \) denote a shingle generated for some vertex during the \( j \)th 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 \( V_t \), 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 \( G_1(S_1, V_r', E') \) in its adjacency list form, such that \( S_1 \) represents the set of distinct shingles generated during this phase, and \( V_r' \subseteq V_r \) represents the subset of vertices that contributed to at least one shingle. Therefore, the output of this phase is \( G_1 \). We call the shingles in \( S_1 \) _first level shingles_.

**Shingling Phase II:** Using \( G_1 \) 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 runtime 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 \(<\text{key}, \text{value}>\) 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 \( u \in V_t \) 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 \([c \times (c \times |\Gamma(u)|)]\).

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
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 \( T_1 \ldots T_c \). Each \( T_j \) keeps track of the minimum \( s \) elements seen so far for that trial. When a new \( <u, v> \) tuple is streamed in, the element \( v \) 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 $O\left(\frac{m \times c \times s}{p_m}\right)$. As for the reducers, assuming a balanced distribution of the grouped tuples, each reducer is expected to take $O\left(\frac{m \times c \times s}{p_r}\right)$ time. The overall run-time complexity for Phase I is expected to be $O\left(\frac{m \times c \times s}{p_m} + T_{\text{group}}(m \times c) + \frac{m \times c \times s}{p_r}\right)$, where $T_{\text{group}}(m \times c)$ represents the time for the MapReduce framework to group $m \times c$ tuples. In practice, assuming the graph is well connected, this run-time can be expected to be significantly smaller than the theoretical bound.

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 $O\left(\frac{n \times c^2}{p_m} + T_{\text{group}}(n \times c^2) + \frac{n \times c^2 \times s}{p_r}\right)$ 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 their algorithm and ours 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]).

<table>
<thead>
<tr>
<th>Input label</th>
<th>Number of edges</th>
<th># proteins</th>
<th># domains</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>11M</td>
<td>11,416,776</td>
<td>8,407,839</td>
<td>11,823</td>
<td>8,419,662</td>
</tr>
<tr>
<td>8M</td>
<td>8,388,608</td>
<td>6,639,087</td>
<td>3,537</td>
<td>6,642,624</td>
</tr>
<tr>
<td>4M</td>
<td>4,194,304</td>
<td>3,681,067</td>
<td>665</td>
<td>3,681,732</td>
</tr>
</tbody>
</table>

**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
Input: Edge list \((m \text{ edges})\)
Output: \(L(0)\)

Map()

Input: \(<u, v>\)
if \(u < v\) then
  emit \(<u, v>\)
else
  emit \(<u, u>\)
end if

Reduce()

Input: \(<u, \text{list of values}>\)
Let \(v_j = \arg \max v\)
emit \(<u, v_j, "c">\)

Let \(L(0) = \{<u, v, "c"> | \forall u \in V(G)\}\)

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

**UpdateLabels(i):**

Input: \([\text{Edge list } (m \text{ edges})]\) concatenated with \(L(i - 1)\)
Output: \(L(i)\)

Map()

Input: \(<u, v, \text{tag}>\>
if tag is empty then
  emit \(<v, 1>, u >\)
else
  emit \(<u, 0>, v >\)
end if

Reduce()

Input: \(<u, \text{list of values}>\>
Let \(u_j \leftarrow v\), for \(<u, 0>, v >\>
emit \(<u, u_j>\>
\forall <u, 1>, v >:\)
  emit \(<u, v_j>\>

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

**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 \(c = 100\).

<table>
<thead>
<tr>
<th>Number of map tasks and reduce tasks</th>
<th>4M</th>
<th>8M</th>
<th>11M</th>
</tr>
</thead>
<tbody>
<tr>
<td>((p_m = 16, p_r = 11))</td>
<td>2,188</td>
<td>4,397</td>
<td>6,874</td>
</tr>
<tr>
<td>((p_m = 32, p_r = 11))</td>
<td>985</td>
<td>1,962</td>
<td>3,599</td>
</tr>
<tr>
<td>((p_m = 64, p_r = 11))</td>
<td>684</td>
<td>1,223</td>
<td>1,701</td>
</tr>
<tr>
<td>((p_m = 128, p_r = 37))</td>
<td>531</td>
<td>1,740</td>
<td>1,241</td>
</tr>
</tbody>
</table>

Efficiency

Efficiency

<table>
<thead>
<tr>
<th>Number of map tasks</th>
<th>Ideal</th>
<th>(m=4M)</th>
<th>(m=8M)</th>
<th>(m=11M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>100 %</td>
<td>100 %</td>
<td>100 %</td>
<td>100 %</td>
</tr>
<tr>
<td>32</td>
<td>96 %</td>
<td>95 %</td>
<td>95 %</td>
<td>95 %</td>
</tr>
<tr>
<td>64</td>
<td>92 %</td>
<td>91 %</td>
<td>91 %</td>
<td>91 %</td>
</tr>
<tr>
<td>128</td>
<td>88 %</td>
<td>87 %</td>
<td>87 %</td>
<td>87 %</td>
</tr>
</tbody>
</table>

Fig. 4. Parallel efficiency of Shingling Phase I.

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.

V. CONCLUSIONS

In this paper, we presented a novel MapReduce algorithm called \(p\text{Clust-mr}\) that can be used to identify dense subgraphs...
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

