Parallel algorithms for clustering biological graphs on distributed and shared memory architectures

Inna Rytsareva

School of Electrical Engineering and Computer Science,
Washington State University,
Pullman, WA 99164, USA
E-mail: inna.rytsareva@email.wsu.edu

Timothy Chapman

Jack Baskin School of Engineering,
University of California,
Santa Cruz, CA 95064, USA
E-mail: rsog412@gmail.com

Ananth Kalyanaraman*

School of Electrical Engineering and Computer Science,
Washington State University,
Pullman, WA 99164, USA
E-mail: ananth@eecs.wsu.edu

Abstract: Graph algorithms on parallel architectures present an interesting case study for irregular applications. In this paper, we address one such irregular application — one of clustering real world graphs constructed out of biological data using parallel computers. While theoretical formulations of the clustering operation are either intractable or computationally prohibitive, efficient heuristics exist to tackle the problem in practice. Yet, implementing these heuristics under a parallel setting becomes a significant challenge owing to a combination of factors including: irregular data access and movement patterns, dependence of computational workload on the input, and a general need to maintain auxiliary pointer-based data structures. In this paper, we present the design and evaluation of two different parallel implementations of a popular serial graph clustering heuristic called the *Shingling heuristic*, which was originally developed by Gibson *et al.* Our first implementation, called *pClust-sm*, is an OpenMP algorithm that targets shared memory multicore platforms. Our second implementation, called *pClust-mr*, targets distributed memory clusters running Hadoop MapReduce. Even though both implement the same serial algorithm, their underlying implementations are vastly different owing to the differences in their target platforms and programming environments. In the shared memory implementation, we were able to improve both the asymptotic runtime and memory complexities of the serial implementation, and drastically reduce the time to solution from the order of several days to a few minutes on larger inputs ($\sim$100M edges). We evaluated the performance on two different shared memory platforms — a commodity large memory (8-core, 32 GB) compute node, and a single node of a specialized SGI Altix UV. With the Hadoop MapReduce implementation, while we were able to demonstrate linear scaling up to 64 cores on modest sized inputs ($\sim$11M edges), the runtimes were between 1-2 orders of magnitude larger compared to the shared memory implementation. Yet this was sufficient to enhance the problem size reach by about two orders of magnitude relative to a previous serial (single-threaded) implementation, in roughly the same amount of time.

Keywords: Graph clustering; shared memory OpenMP algorithm; MapReduce algorithm; hash tables; union-find data structure; Shingling heuristic; protein family identification; protein domain family.

Reference to this paper should be made as follows: Rytsareva, I., Chapman, T. and Kalyanaraman, A. (2012) 'Parallel algorithms for clustering biological graphs on distributed and shared memory architectures', *Int. J. High Performance Computing and Networking*, Vol. 1, Nos. 00, pp.1–15.

1 Introduction

Biological data, both naturally occurring and synthetically generated, lend themselves well to graph-based representations. Vertices can be used to represent experimentally acquired data and edges (weighted or unweighted, directed or undirected) can be used to represent pairwise relationships between the 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, the clustering formulation targeted in this paper is as follows: Given an input graph $G(V, E)$ with n vertices and m edges, "*clustering*" is the act of grouping vertices into tight-knit clusters, where the members of each cluster are closely related to most (if not all) other members of the same cluster, and sparsely related (if at all) to the members of other clusters. A scalable clustering solution that models this formulation can be useful to a number of applications within computational biology. For instance: it can be used to reduce redundancy within sequence repositories; identify complexes within metabolic networks (3); identify core groups of proteins that constitute a protein family (37; 43; 46) and in the process also help assign family memberships for newly found peptide candidates (46); help in the construction of mass spectral libraries for peptides (25); and can be used to condense the space of plausible computer-generated phylogenetic trees (31).

In standard literature, the above clustering formulation is also sometimes referred to as *dense subgraph detection*. It is to be noted that this is subtly different from community detection, and from graph partitioning to a larger degree. In the case of graph partitioning, vertices are partitioned into a pre-specified number of roughly equal-sized groups. In clustering though, clusters are allowed to have different sizes, and the number of clusters and their size distribution are both unknown at input. While this is also the case with community detection, such methods generally do not allow for control over the densities of the output communities. Instead their objective function is to optimize the overall clustering modularity (36) — a related measure which seeks to identify natural divisions that exist in a given network.

Despite its potential to address a broad range of problems, the 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., (46)). 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 variable-sized dense subgraphs (or quasi-cliques), and theoretically speaking, several of the corresponding optimization problems are computationally hard problems (1; 17; 24) or with large degree polynomial methods (35; 37). 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* (19). Posed as a dense subgraph detection problem for web community detection, 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 (43). Put briefly, this approach, called *pClust*, transforms the problem into one of bipartite graph clustering so that the approach developed by Gibson *et al.* can be used. The results (43; 42) 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 its 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 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.

1.1 Contributions

In this paper, we present two parallel algorithms for the Shingling heuristic targeted at two different architectures: one using OpenMP multithreading for symmetric multiprocessor nodes with a shared memory, and another using Hadoop MapReduce for distributed memory clusters. We chose to evaluate the MapReduce (13) paradigm for distributed memory not only because it is rapidly becoming a popular standard for data-intensive applications but also because the fundamental operations required to implement the Shingling heuristic naturally lend themselves to a map-reduce structure. The contributions can be summarized as follows:

- **pClust-sm:** In our OpenMP implementation, we were able to reduce both the runtime and memory requirements of our previous serial implementation by taking advantage of the shared address space among cores. This is achieved by replacing a sorting step with hash tables, and then using an on-the-fly generation scheme for reporting clusters using union-find data structure, which reduces the peak memory usage by a large constant factor (≈ 100x in practice). Results show that *pClust-sm* scales appreciably up to 8 cores for larger inputs on a single node of an SGI Altix UV shared memory machine and another commodity 8-core Linux node. More importantly, it has allowed us to directly solve a real world protein sequence homology graph with 1.2 million vertices (100M edges), in just over 4 minutes using 8 cores. This processing time using *pClust-sm* is significantly less than our previous processing time obtained using *pClust*, which took about 30 minutes to cluster all 65K connected components on a 128-core cluster (42).

- **pClust-mr:** Our distributed memory implementation has been designed as a pipelined, multi-stage MapReduce implementation to efficiently extract parallelism from the MapReduce framework. The underlying algorithm transforms the operations of the Shingling heuristic into a combination of standard MapReduce primitives such as *map*, *reduce* and *group/sort*. Experimental 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). However, the runtimes were between 1-2 orders of magnitude larger compared to the shared memory implementation. Yet, relative our serial implementation *pClust*, this was sufficient to enhance the problem size reach by about two orders of magnitude ($\times 10^4$ to $\times 10^6$ vertices) in roughly the same amount of time.

Even though the two parallel algorithms implement the same serial heuristic, there are significant differences in their algorithmic structure, data structures used and in the techniques deployed, owing to the differences of their respective target architectures. Consequently, the study serves as a way to characterize the two architectures (shared memory and Hadoop-based distributed memory) for the irregular problem of graph clustering.

2 Background and Related Work

The problem of finding a maximum dense subgraph within an input graph is solvable in polynomial time (10; 20; 27). 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; 17; 24). Our problem represents a more generalized version of these variants, wherein the goal is to find multiple, variable-sized 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 the Internet data (e.g., (19)). It turns out that the bipartite graph formulations are also NP-Hard (17; 28).

There is a rich body of clustering related literature in the context of biological applications. A considerable segment is devoted to gene expression/microarray analysis and transcript/genome assembly (reviewed in (14; 15; 22)). For these applications, however, clustering is not generally modeled as a graph problem (except for those that involve string graphs in short read assembly), and simpler agglomerative techniques (e.g., neighbor joining) and single linkage clustering suffice in practice due to the nature of sampling. Such methods also tend to create loose clusters and suffer from error propagation during incremental construction. A different class of applications benefit from a clustering formulation based on detecting denser communities within biological data (e.g., (3; 16; 21; 29; 46)). Brohee *et al.* (7) evaluate clustering techniques for analyzing protein-protein interaction networks. Andreopoulos et al. (2) present a broader, and more conceptual survey of different clustering applications that benefit from tighter cluster techniques.

Independently, in other areas of computing such as social and cyber networks, numerous algorithms have been developed for community detection (reviewed in (26; 28; 34)). M.E.J. Newman, in his pioneering work on discovering community structure from networks (35), 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 (35), 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 (30) for efficiently calculating

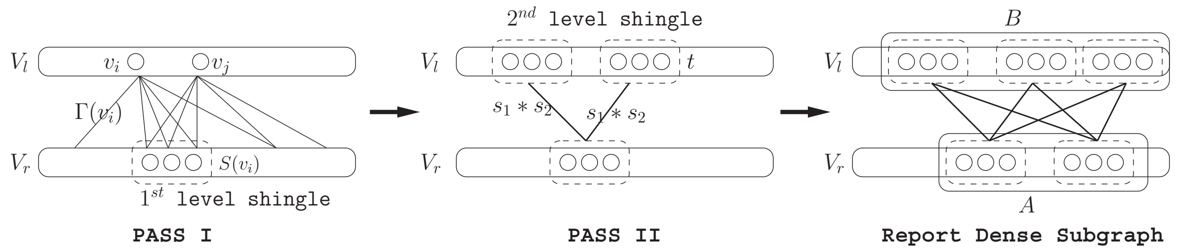

Figure 1 Illustration of the different steps in the Shingling heuristic. Typically, $s_1 = s_2 = s$.

betweenness centrality on graphs. A different approach (37) 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. More recently, Blondel *et al.* (4) developed another community detection method, which is loosely based upon one of Newman's previous modularity-based methods (11). This method, referred to as the Louvian method, has now emerged to be a widely used method for community detection.

Gibson *et al.* developed the Shingling heuristic for identifying dense internet communities (19). The underlying method (described in Section 3.2) uses a bipartite graph approach along with a random sampling procedure and secondary sorting. In (43), 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. Given an input homology graph $G(V, E)$ with n vertices and m edges, *pClust* implementation's runtime is dominated by sorting step which sorts $O(n \times c)$ values, where c is a parameter (typically ≥ 100); and its memory complexity is $O(n \times c^2)$.

The shared memory implementation proposed in this paper reduces this runtime through the use of a hash table and by parallelizing under the OpenMP model; it also reduces the memory complexity to $O(n \times c)$. The MapReduce-based implementation, on the other hand, maps this problem to distributed memory Hadoop clusters and runs with a disk storage requirement of $O(m + n)$, where m is the number of edges.

3 Methods

3.1 Problem definition

Let $G = (V_l, V_r, E)$ denote an undirected bipartite graph[1] with n vertices ($n = |V_l| + |V_r|$) and m edges. Let $n_l = |V_l|$ and $n_r = |V_r|$. Let $\Gamma(u) = \{v \mid (u, v) \in E\}$ denote the set of *links* for vertex u. As a convention, we will use u to denote a vertex from V_l and v to denote a vertex from V_r.

Defined loosely, a *dense subgraph* in a bipartite graph is a 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 links in V_r'.

Given $G(V_l, V_r, E)$, our goal is to find a set of variable-sized dense subgraphs within G. Although one could associate a notion of maximality for this problem definition, wherein each group identified cannot be expanded with more vertices without becoming less dense, we avoid such a definition because that would necessitate a fixed density-cutoff and the heuristic we implement does not use such a cutoff.

3.2 The serial algorithm

Definition 3.1: Given an integer constant s, a "shingle" of a vertex u (6) is defined as an arbitrary $s-$element subset of $\Gamma(u)$.

If two vertices are part of a dense subgraph, they can be expected to share a large fraction of their neighbors in common. Therefore, a brute-force way to detect vertices that are part of the same 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 quadratic search space. More specifically, it obtains random samples of size s (called shingles) 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 links and hence with a high probability are also expected to share a shingle (5). 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 links is achieved as follows (19): First, c random permutations of the vertices in $\Gamma(u)$ are obtained using a fixed set of c random number pairs $\{<A_j, B_j> | j \in [1, c]\}$. The top s elements within each permutation are then said to represent a *shingle*. The random permutation of $\Gamma(u)$ for a given random trial $j \in [1 \ldots c]$ is obtained as follows: Assume that every $v \in \Gamma(u)$ is associated with a unique integer id. Then a bijection from the set

$\Gamma(u)$ to a new set $\Gamma^j(u)$ is computed by taking every $v \in \Gamma(u)$ and mapping it to an element $v^j \in \Gamma^j(u)$ such that $v^j = (A_j \times v + B_j)\%P$, where P is a big prime number. Consequently, sorting $\Gamma^j(u)$ yields the random permutation for $\Gamma(u)$ for trial j. A permutation thus obtained preserves the min-wise indepedent property that guarantees, with high probability, that vertices of a densely connected subgraph would also share a shingle (5; 6; 19).

The above heuristic is implemented in *pClust* (43) in three phases (also see Figure 1):

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. In our implementation, the sorting required to generate a shingle from $\Gamma^j(u)$ is implemented by performing an on-the-fly enumeration of $\Gamma^j(u)$ and alongside keeping track of an s-sized array that records the minimum s elements at any point of time through a simple insertion sort. The small values of s expected to be used in practice (typically under 10) justify a simple insertion sort-based approach. 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_l, a sorting is done to gather all vertices that generated each shingle. This shingle is done once for each random trial (so that shingles from different trials do not get mixed). 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:

- For Shingling Phase I, all edges are traversed c times to generate $\Gamma^j(.)$ from all vertices, and the subsequent insertion sort to derive the shingles entails s comparisons per edge form in every $\Gamma^j(.)$. One global sort of all shingles produced for each trial is performed using quick sort. Note that the phase outputs a total of $n_l \times c$ output shingles (although not necessarily distinct) across all trials. This yields a run-time complexity of $O(m \times c \times s + c \times T_{sort}(n_l))$. It is worth noting that if S_1 denotes the set of first-level shingles, then $c \leq |S_1| \leq n_l \times c$, and the denser the input graph is, the smaller the value of $|S_1|$ tends to be;
- Similarly, for Shingling Phase II, the runtime complexity is $O(|E'| \times c \times s + c \times T_{sort}(|S_1|))$, where E' is the set of edges in G_I output by phase I; note that $|E'| \leq n_l \times c$ (or more precisely, equal to $|V_l'| \times c$). And if S_2 denotes the set of second-level shingles then $c \leq |S_2| \leq |S_1| \times c$.
- 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 $O(max\{m + n, |E'|, |E''|\})$.

3.2.1 Parallelization methodology

The first step in each of the shingling phases, which is to generate c shingles per vertex, is a data-parallel step and can be parallelized by distributing the vertices along with their adjacency lists in parallel so that each parallel task can independently compute the shingle set corresponding to the vertices it owns. However, under this approach it would become important to keep the workload across parallel tasks balanced factoring in the variability of adjacency list lengths. An alternative strategy is to operate using an edge-list representation. Note that the subtask of generating the random permutation involves c independent traversals per edge to generate the mapping from Γ to Γ^j $(\forall j \leq c)$, which can be parallelized. Subsequently, the next subtask of picking the minimum s elements from each Γ^j becomes one of gathering the mapped entries by source vertex and trial number, which can also be parallelized using sort or grouping functions. We explore both approaches — the former in our shared memory implementation and the latter in our MapReduce implementation. The final phase of generating connected components can be implemented as a parallel breadth first search routine.

3.3 pClust-sm: A shared memory parallel algorithm

For the design of *pClust-sm*, we set off with the goals of improving both run-time and memory complexities.

The main algorithmic steps in *pClust-sm* are shown in Algorithm 1 and can be described as follows:

First, the input graph is loaded by the master thread from I/O into the main memory in the form of an adjacency list (i.e., edge list for one vertex per line). Subsequently, the (n) vertices are dynamically distributed in parallel (in batch sizes of 64) to individual threads. For each vertex u_i, its owner thread generates c shingles. Recall that the shingle generation function randomly permutes $\Gamma(u_i)$, sorts them, and then selects the top s elements. The *label* of a shingle is the string obtained by concatenating the labels of its top s elements (in that order). Each shingle is also mapped to an integer ID using a hash function, and to ensure uniqueness, the combination <ID,label> is used to identify a shingle. To store the generated list of first level shingles, *pClust-sm* uses a hash table H that is shared among all the threads[2]. By using a hash table, the algorithm groups together all the vertices generating a given shingle, thereby eliminating the need for an explicit sorting step in Phase I. However, the key is to implement the hash table efficiently as multiple threads are trying to insert shingles concurrently.

Algorithm 1 pClust-sm(Input: $< G(V_l, V_r, , E), s, c >$)

```
Let $id = omp\_get\_thread\_num()$;
Init: $S^{id} \leftarrow null$;
Init: Hash Table $H$ with $n \times c$ entries;
/*Phase I*/
#pragma omp parallel default(shared)
for ($i = 0$; $i < n_l$; $i$++) do
    Let $u_i = i^{th}$ vertex in $V_l$;
    for ($j = 0$; $j < c$; $j$++) do
        $s \leftarrow$ Generate $j^{th}$ shingle from $\Gamma(u_i)$;
        Insert($< s, u_i >$) into $H$;
        Append inserted location for $s$ to $S^{id}$;
    end for
end for
/*Combined Phases II and III*/
Init: UnionFind UF[$1 \dots n$];
#pragma omp parallel default(shared)
Init: $k = 0$; $next \leftarrow S^{id}[k$++$]$;
while next!=null do
    Let $s \leftarrow$ shingle object pointed to by $next$;
    for ($j = 0$; $j < c$; $j$++) do
        $t(s) \leftarrow$ Generate $j^{th}$ shingle from $\Gamma(s)$
        Update $UF \leftarrow \{\bigcup_{\forall v_i \in s} v_i\} \bigcup \{\bigcup_{\forall u_i \in t(s)} u_i\}$
        Let $next \leftarrow S^{id}[k$++$]$;
    end for
end while
Output UF clusters in serial.
```

Our design of the hash table is as follows: We use a hash function to map the shingle to a hash index, and use chaining to resolve collision. Algorithm 2 shows in detail the steps involved while inserting a shingle object into the hash table. Note that a shingle object to be inserted is of the form $< s, u >$, where s denotes the shingle identifier and u denotes the vertex that just generated it. Basically, the idea is to lock at the resolution of a hash index, and to defer locking of an index as late as possible so as to keep the time an index locked minimal. The index mapping to a shingle is first probed — here, two scenarios are possible: an entry corresponding to this shingle is either already there in that index, or it is not there. In the former case, all that is required is to append the vertex that generated this shingle to its neighbor list; this requires locking only the shingle object. In the latter case, the shingle object is appended to the list maintained at the index and then the vertex is added to its neighbor list; this requires locking the entire hash index.

Algorithm 2 HashTable-Insert(Input: $u, \{v_1, v_2 \dots v_s\}$)

```
Let $key \leftarrow hash(“v_1\#v_2\# \dots \#v_s'') \% \ tablesize$;
Let $fetch \leftarrow$ Fetch the handle to the hash entry for $key$;
if (fetch) then
    Lock($shingle\_node\_at\_H[key]$);
    Append $u$ to neighbor list of the shingle;
    Unlock($shingle\_node\_at\_H[key]$);
else
    Allocate node for shingle;
    $fetch \leftarrow$ Fetch again the handle to the hash entry for $key$;
    if (fetch) then
        Deallocate shingle node;
    else
        Lock($H[key]$);
        Append new shingle node to hash entry at $key$;
        Unlock($H[key]$);
    end if
    Lock($shingle\_node\_at\_H[key]$));
    Append $u$ to neighbor list of the shingle;
    Unlock($shingle\_node\_at\_H[key]$);
end if
```

Figure 2 illustrates Phase I.

The original serial algorithm *pClust* implements Shingling Phase II and the Phase III (generating second level shingles and enumerating connected components) as two different phases. Because this requires that all second level shingles be stored in memory before proceeding to the next phase, this leads to a peak memory usage of $O(n \times c^2)$, as that is the upperbound on the number of distinct second level shingles. Given that the value of c is expected to be large (≥ 100), this becomes a significant memory bottleneck in practice.

In *pClust-sm*, we completely eliminate the need to store second level shingles and thereby improve the memory complexity to $O(n \times c)$ (space required to store the first level shingles). This is achieved based on the following observation: Since the output clustering is defined by the connected components formed by first level shingle to second level shingle connections. Therefore, as soon as a second level shingle is generated from a first level shingle, the constituent $2 \times s$ vertices

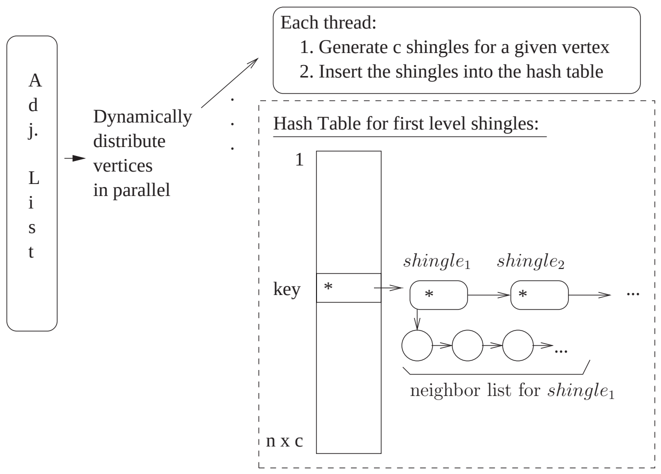
Each thread:
1. Generate c shingles for a given vertex
2. Insert the shingles into the hash table
Adj. List
Dynamically distribute vertices in parallel
Hash Table for first level shingles:
1
key
n x c
$shingle_1$
$shingle_2$
neighbor list for $shingle_1$

Figure 2 *pClust-sm*: Illustration of Phase I and the hash table shared by all threads in *pClust-sm*. Asterisks indicate the resolution at which locks could be potentially placed while inserting.

of those two shingles can be merged into one set, thereby eliminating the need to explicitly store all the second level shingles.

The above idea basically implies that the Phases II and III can be combined to execute in tandem. The second half of the Algorithm 1 illustrates this approach. Merging of vertex sets is performed using the classic union-find data structure. There are two ways to parallelize the process of generating second level shingles from the list of first level shingles: one option is to distribute the list of nonempty hash table entries dynamically to the the threads, so that each thread can generate second level shingles for its internal linked list of first level shingles. In our current implementation, we follow a simpler approach: the thread that created the first instance of a shingle object owns the responsibility of generating second level shingles for that object. This is implemented by keeping track of one linked list per thread during Phase I and then navigating them using this phase. This simpler approach runs the potential risk of creating load imbalance situations (there could be a skew in the number of shingle objects owned by a thread). Interestingly, our experimental results, at least for the inputs tested, did not result in such an imbalance.

Algorithm 3 shows the algorithm to perform the union/merge operation using the union-find data structure. Basically, *Union* uses a union-by-rank heuristic and the *Find* operation is implemented using path compression. Locking is performed at the union-find array entry which is being updated. Due to the deferred locking scheme, it is possible that an entry, between the time it is queried for its parent and the time the lock is placed, is updated by a second thread. Even though this situation is likely to be rare, we handle this by skipping the current attempt at union and re-attempting until no other threads have updated the entry. In our experiments, we never encountered re-attempts.

Algorithm 3 Union (Input: a, b)

```
$done \leftarrow$ false;
repeat
    Let $root_a \leftarrow Find(a)$;
    Let $root_b \leftarrow Find(b)$;
    if ($root_a == root_b$) then
        return;
    end if
    if ($UF[root_a].rank < UF[root_b].rank$) then
        Lock($UF[root_a]$);
        if ($UF[root_a].parent$ or $UF[root_b].parent$ is not null) then
            Unlock($UF[root_a]$);
            continue;
        else
            $UF[root_a].parent = root_b$;
            Update $UF[root_a].rank$;
            Unlock($UF[root_a]$);
            $done \leftarrow$ true;
        end if
    else
        SwapLabels(a,b);
    end if
until done
```

3.3.1 Analysis

Phase I: In the current implementation, we initialize the hash table with size equal to $n_l \times c$ as that is the upperbound on the number of distinct first level shingles. This makes the memory complexity for Phase I $O(n_l \times c)$. As for the runtime complexity, the worst-case runtime for Phase I is $O(\frac{m \times c \times s}{p})$, where p is the number of OpenMP threads (assuming amortized constant time per hash table insert and an optimistic linear scaling behavior). Note the conspicuous absence of the sorting time.

Combined Phases II and III: The worst-case runtime for generating second level shingles is $O(\frac{|E'| \times c \times s}{p})$ and the time to perform the merging using union-find is expected to take $O(\frac{(|S_1|+|S_2|) \times s \times \alpha(n)}{p})$. As for the memory complexity, the memory requirement for the union-find data structure is $O(n)$ which is strictly dominated by the space requirement in Phase I.

3.3.2 Implementation

The program *pClust-sm* was written in C++ and OpenMP. In the current implementation, the constant of proportionality in the memory complexity $O(n_l \times c)$ is 40. A beta testing version of the code is available as an open source under GNU Lesser GPL license at `http://code.google.com/p/pclust-sharedmem/`.

3.4 pClust-mr: A MapReduce algorithm

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.

3.4.1 Shingling Phases I and II

Design challenges: The serial algorithm assumes an adjacency list format. This provides direct and easy access to all of the 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 viewed as an atomic unit, this assumes that there is sufficient memory on the mappers to store at least one line of the input. And since in the worst-case, a vertex's adjacency list could be $\Theta(n)$, it 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 make the line lengths uniform, but becomes too expensive in 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.

Algorithm: We devised an algorithm that overcomes all the above outlined challenges. Our algorithm is illustrated in Figure 3. 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 every link from u, 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 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 $<u, v^j>$ 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 is identified as the *Unique I* phase in Figure 3. 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.

3.4.2 Analysis of the Shingling phases

The disk storage complexity is $\Theta(m)$, because the algorithm operates using an 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.

Phase II's analysis is similar except that the input number of edges in G_I is bounded by $O(n_l \times c)$ (worst-case represented by a sparse graph). Therefore,

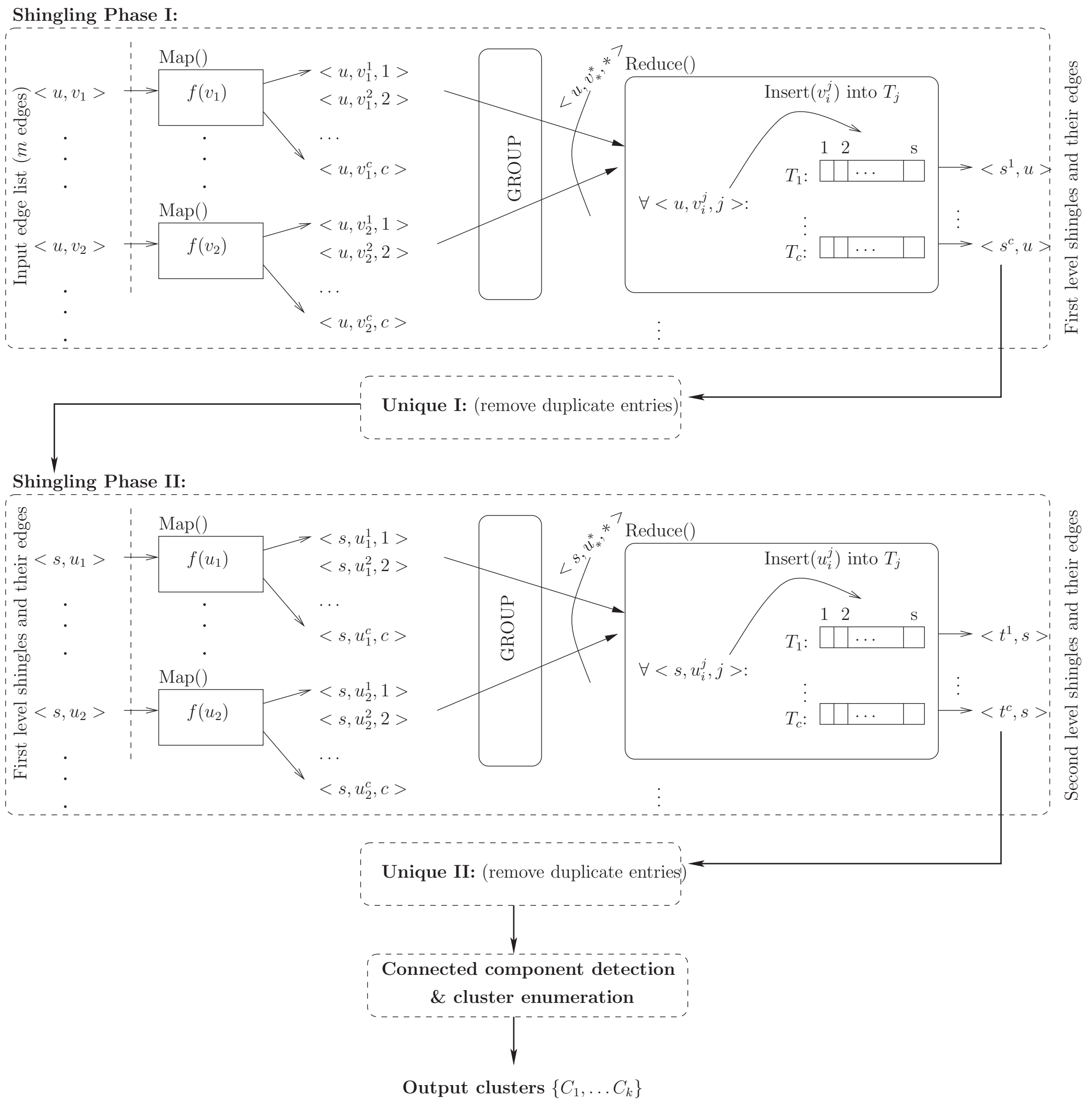

Figure 3 Illustration of the *pClust-mr* algorithm. The figure shows the detailed map-reduce stages for Shingling Phases I and II. The function $f()$ (inside Map()) takes as argument a vertex v in the form of its integer id and generates c forms of v, viz. $\{v^1, v^2, \ldots v^c\}$; where $v^j = (A_j \times v + B_j)\%P$, $j \in [1 \ldots c]$ corresponds to the j^{th} trial number, $< A_j, B_j >$ denote the j^{th} 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 tuples of the form $< u, \ldots >$, the intermediate key is field u.

a run-time complexity of $O(\frac{n_l \times c^2}{p_m} + T_{group}(n_l \times c^2) + \frac{n_l \times c^2 \times s}{p_r})$ follows. In practice, graphs are expected to be connected enough to imply significantly smaller than the theoretical bound.

3.4.3 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 (45). The MapReduce algorithm is illustrated in detail in Figure 4. 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 there is a nearly-identical algorithm reported by Kang *et al.* (23). The only difference between our algorithm and their's is

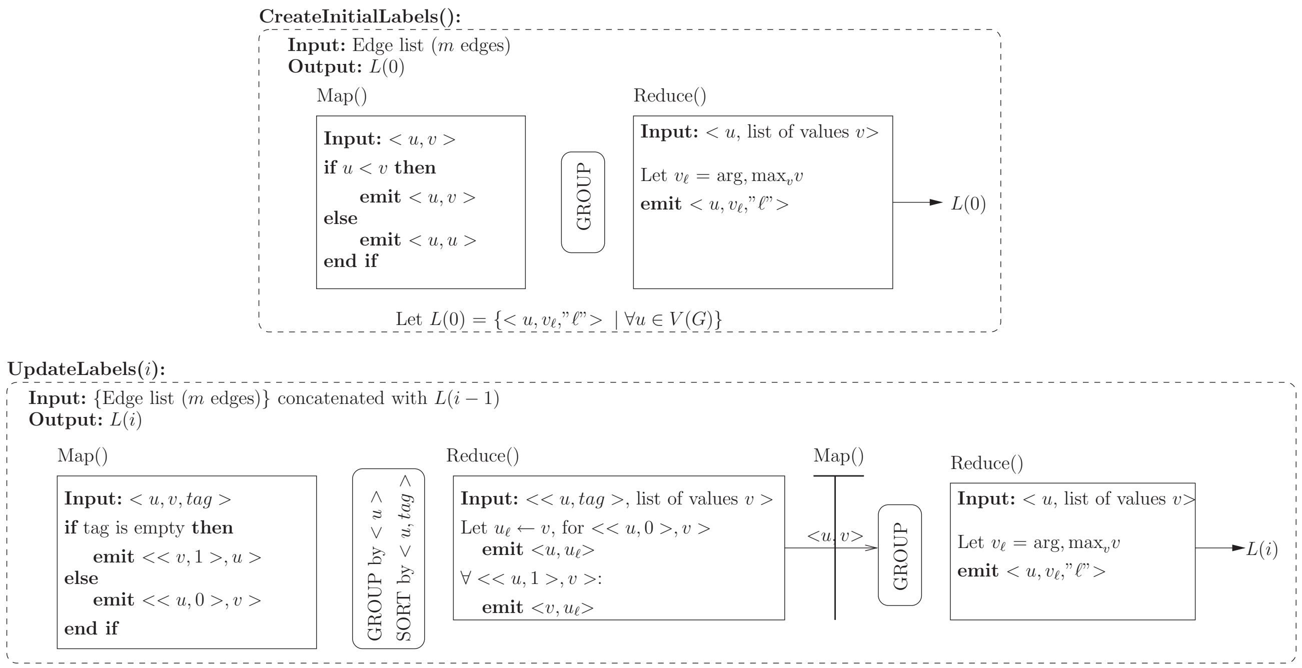
CreateInitialLabels():
Input: Edge list (m edges)
Output: $L(0)$
Map()
Input: $< u, v >$
if $u < v$ then
emit $< u, v >$
else
emit $< u, u >$
end if
GROUP
Reduce()
Input: $< u$, list of values $v>$
Let $v_\ell = \arg, \max_v v$
emit $< u, v_\ell, "\ell" >$
$L(0)$
Let $L(0) = \{< u, v_\ell, "\ell" > \mid \forall u \in V(G)\}$
UpdateLabels(i):
Input: {Edge list (m edges)} concatenated with $L(i-1)$
Output: $L(i)$
Map()
Input: $< u, v, tag >$
if tag is empty then
emit $<< v, 1 >, u >$
else
emit $<< u, 0 >, v >$
end if
GROUP by $< u >$
SORT by $< u, tag >$
Reduce()
Input: $<< u, tag >$, list of values $v >$
Let $u_\ell \leftarrow v$, for $<< u, 0 >, v >$
emit $<u, u_\ell>$
$\forall << u, 1 >, v >$:
emit $<v, u_\ell>$
Map()
$<u, v>$
GROUP
Reduce()
Input: $< u$, list of values $v>$
Let $v_\ell = \arg, \max_v v$
emit $< u, v_\ell, "\ell" >$
$L(i)$

Figure 4 *pClust-mr*: Illustration of the MapReduce algorithm for connected component detection. The algorithm terminates when $L(i) == L(i-1)$.

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 (23)).

3.4.4 Implementation

The program *pClust-mr* was written in Java combined with Hadoop MapReduce API calls. A beta testing version of the code can be obtained by contacting the authors.

4 Experimental Results

4.1 Experimental setup

Test Platforms: For *pClust-sm*, we used two test platforms: i) A *commodity node*, which is an 8-core (Intel Xeon(R) 2.50GHz CPUs) 32 GB RAM compute node running CentOS 5.7 Linux, and ii) the NICS *Nautilus* supercomputer, which is an SGI Altix UV 1000 supercomputer with a total of 1,024 Intel Nehalem EX (6-core) processors, 4 TB of shared memory along with a 427 TB Lustre file system. Each node of Nautilus contains two Nehalem processors for a total of 12 cores, and share 16 GB RAM. For our experiments, we tested up to 16 threads. We used the SGI command *omplace* to ensure the placement of successive threads on unique CPUs.

For *pClust-mr*, 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.

Input data: We used two types of biological graphs for testing our methods (see Table 1). The first input graph is a sequence homology graph constructed for a set of 1.28×10^6 amino acid sequences (43). The sequence data were originally downloaded from the CAMERA (8) portal and represent a collection of open reading frames extracted from an ocean metagenomics project (46). The homology graph, constructed using the pGraph method (43; 44), contains 1.28×10^6 vertices which correspond to the amino acid sequences and 100×10^6 edges which correspond to those pairs of sequences which are homologous. Clustering this graph would yield core members of protein families (46; 43). We used this set as it is a large input and also the answers can be matched for correctness and performance improvements over our earlier serial implementation of clustering (43). For scalability studies, smaller subgraphs were extracted from this large graph containing approximately 50M, 30M and 20M edges.

The second graph is a bipartite graph that we constructed after downloading the entire *Pfam-A* protein-domain database (18; 38). This database contains a set of proteins and a set of domains contained in those proteins. Each protein contains only a small subset of domains, and a domain can be contained in multiple proteins. Therefore, this data automatically renders itself in the form of a bipartite graph, where edges are between a protein and a domain if that protein contains that domain. Applying our clustering method to this input would yield a novel way to characterize proteins into "families" based on multiple conserved domains. The entire Pfam-A graph contains 8,419,662 vertices (= 8,407,839 proteins + 11,823 domains), and 11,416,776 edges. Smaller subgraphs were extracted from this large graph for scalability studies. A value of $s = 2$ and $c = 100$ was used in all our experiments, based on our earlier experiments with *pClust* (43; 42).

4.2 Performance results

4.2.1 Performance of pClust-sm

We performed extensive analysis of *pClust-sm* on the CAMERA input data sets. Table 2 shows the runtime of *pClust-sm* as a function of the input size on up to 16 cores of the Nautilus supercomputer. The loading phase is a serial step performed by the master thread. It can be seen that the net runtime for the remaining phases (Phases I, II and III which are the parallel part) increases proportional to the input size with the exception of 20M to 30M; the latter is because the actual computational work in Phase I, which is expected to be the dominant phase, is determined more by the degree distribution and the neighbor list composition (to pick top s elements) than the size of the graph.

Table 2 also shows *pClust-sm*'s runtime as a function of the number of threads. It can be observed that as the input size grows the scaling also improves, with the best scaling obtained for the largest input (CAMERA-100M). For smaller input sizes while there is still some performance improvement with the addition of cores, but the scaling is not linear owing to the combination of reduced work and a relative increase in overhead due to locks. Figure 5 shows the speedup of *pClust-sm*. The results show that even the largest input data containing 1.2M vertices is not large enough to benefit beyond 8 threads. The rapid deterioration of speedup from 8 to 16 threads is probably because there are only 12 cores per board and the system starts to use the network interconnect to access the shared memory.

The results also show that the time to cluster the largest input graph with 1.2M vertices takes just over 4 minutes (249 seconds) on 8 threads. This is a substantial improvement over the previous serial version of the *pClust* code, which took 64 CPU hours (43). Put another

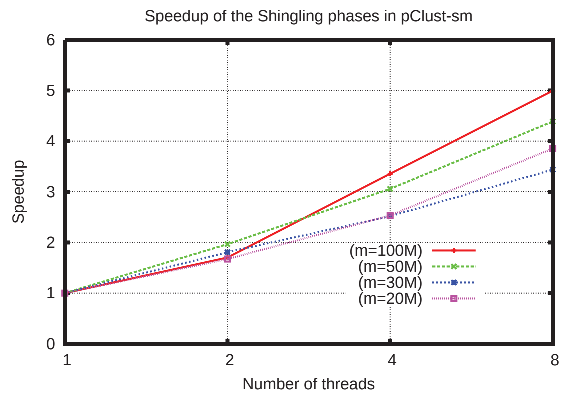

Figure 5 *pClust-sm*: Speedup for up to 8 cores on Nautilus.

way, our new shared memory implementation operates on the input graph directly and reduces the time to solution for 1.2M vertices input from 64 CPU hours to 33 CPU minutes (i.e., 249s on 8 threads).

To determine performance changes owing to architectural differences, we also ran *pClust-sm* on the same CAMERA input on the 8-core Linux commodity node. Table 3 shows the phase-wise breakdown of the runtime for the CAMERA-100M input graph. We exclude the loading phase from this analysis because it is a serial step.

As can be expected, Phase I is the dominant phase, and it scales linearly up to 4 threads after which the scaling gradually deteriorates at 8 threads on both platforms (e.g., 236s to 163s on Nautilus). After incorporating more timing information on the Nautilus platform, we found that the time to fetch the handle to the hash table index (which does not involve locking) reduced by only 40% (instead of 50%) from 4 to 8 threads; and the time to do the insertion which includes the locking time reduced by 35%. However, the portion of runtime spent obtaining the lock itself did *not* increase from 4 to 8 threads (in fact, it remained almost flat between 7-8% for all thread counts 2 to 8). In addition, the number of hash table collisions was practically negligible for all thread counts tested ($\geq 2,000$ out of $\approx$50M insertions). These observations imply that the slowdown in both platforms from 4 to 8 threads must be a result of increased memory access generated by the increased number of threads.

Table 1 Input graph statistics: The first part of the table shows statistics for input graphs constructed from metagenomics protein/open reading frame sequence data downloaded CAMERA. The second part shows the statistics for the graphs constructed from the Pfam-A protein-domain database.

Input label	*Number of edges (m)*	*Number of vertices (n)*	**Input label**	*Number of edges (m)*	*Number of vertices (n)* # *proteins*	# *domains*	*Total*
CAMERA-100M	100×10^6	$1,280 \times 10^3$	**Pfam-A-11M**	11,416,776	8,407,839	11,823	8,419,662
CAMERA-50M	50×10^6	325×10^3	**Pfam-A-8M**	8,388,608	6,639,087	3,537	6,642,624
CAMERA-30M	30×10^6	190×10^3	**Pfam-A-4M**	4,194,304	3,681,067	665	3,681,732
CAMERA-20M	20×10^6	189×10^3					

Table 2 *pClust-sm*: Run-time (in seconds) as a function of input and system sizes on the Nautilus supercomputer. All runs were performed using $s = 3$ and $c = 100$.

Input graph	*Loading phase*	*Time for the remaining phases using p cores*				
		p =1	*p =2*	*p =4*	*p =8*	*p =16*
CAMERA-100M	40	1,044	613	311	209	195
CAMERA-50M	15	474	241	155	108	94
CAMERA-30M	10	282	156	112	82	60
CAMERA-20M	8	266	159	105	69	69

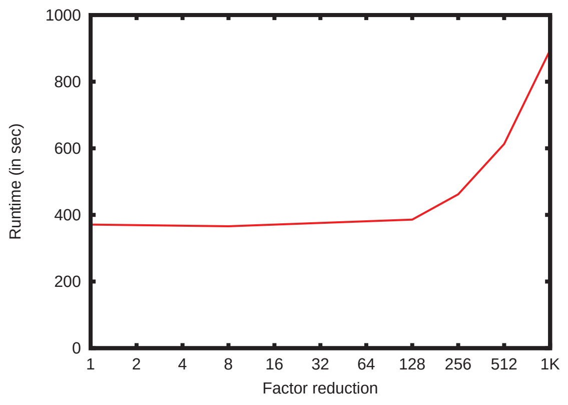

Figure 6 *pClust-sm*: Effect of varying the hash table size on performance. The input was fixed at CAMERA-100M data set with parameters $s = 3$ and $c = 100$.

The only significant difference in the results obtained between the two platforms (see Table 3) is the slowdown during the combined Phases II & III on the commodity node. During this stage, the only data structure that the algorithm heavily operates on is the union-find data structure. Further investigation is needed to identify the cause of this anomaly.

Effect of varying hash table size: For the studies shown above, we initialized the hash table with $n_l \times c$ entries so that it matches the theoretical upperbound on the number of distinct first-level shingles. However, is such a high hash table size required to guarantee reduced collision rates? To answer this question, the impact on performance was studied by reducing hash table size for factors up to 1024x. Figure 6 shows runtime as a function of the factor decrease in hash table size starting at $n_l \times c$, keeping the input fixed at $n_l = 1.28 \times 10^6$ (CAMERA-100M input) and $c = 100$. As can be observed, the hash table size has no effect on runtime up to an 128x factor size reduction, beyond which the performance starts to deteriorate as expected. For instance, reducing the table size from $\frac{n_l \times c}{128}$ to $\frac{n_l \times c}{1024}$ caused in ~2.3x increase in runtime — a manifestation of increased collision rates. The observed point of farthest minima (128x for this input) is data-dependent.

4.2.2 Performance of pClust-mr

We studied the performance of our MapReduce implementation, *pClust-mr*, on the Magellan Hadoop cluster. The numbers of map tasks and reduce tasks are specified at input. Table 4 shows the phase-wise breakdown of the total runtime as a function of the number of map tasks for the Pfam-A-11M input graph. It can be observed that Shingling Phase I is the dominant phase for this input. This is because the transformed graphs (G_I and G_{II}) become orders of magnitude smaller due to the highly sparse nature of the original input graph. Table 4 also shows that *pClust-mr* scales linearly up to 64 map tasks for Shingling Phase I. The lack of scaling for the remaining phases can be attributed to the small sizes of their corresponding input graphs.

While the code exhibits linear scaling at least for Shingling Phase I, the runtimes are substantially higher than the runtimes obtained using our shared memory implementation (*pClust-sm*) for the Pfam-A-11M input (results not shown). For instance, *pClust-sm* took only 192 seconds on 8 cores of the commodity node, while *pClust-mr* took 8,587 seconds on 16 cores of the Hadoop cluster (implying a slowdown factor of roughly ~80x). The larger runtime on the Hadoop cluster is expected because of two factors: i) the I/O bound nature of the Hadoop environment; and ii) the algorithmic level improvements that we were able to implement as part of *pClust-sm* (such as use of hash tables and combining the last two phases) could not be carried over to the distributed memory setting of Hadoop. To quantify these factors, we performed more diagnostics and found the following: Of the 3,599 seconds for Shingling Phase I consumed by *pClust-mr* on the Pfam-A-11M input using 32 cores of the Hadoop cluster, nearly all of it (95.71%) was consumed by the longest running map task[3] (implying a negligible fraction of time needed for the reducers to complete). And the average breakdown for the map task runtime is as follows: 31.34% in local calculations, 47.76% for the grouping of mappers' intermediate key-value pairs, and the remaining 20.90% towards I/O. It is to be noted that even the grouping task that the Hadoop framework carries out is performed in I/O (HDFS). These diagnostics provide a clear picture of the different overheads associated with the Hadoop framework on the clustering application.

In what follows, we present the results of analyzing the scalability of the *pClust-mr* for its time-dominant phase, which is Shingling Phase I. Table 5 shows Shingling Phase I's run-time as a function of the number of map tasks. The results show linear scaling up to 32

Table 3 *pClust-sm*: Breakdown of runtime (in seconds) by phases for the input graph CAMERA-100M on Nautilus and the commodity node.

Phase	*Time using p Nautilus cores*				*Time using p cores of the commodity node*			
	p =1	*p =2*	*p =4*	*p =8*	*p =1*	*p =2*	*p =4*	*p =8*
Phase I	780	467	236	163	1,383	736	417	323
Combined Phases II & III	264	146	75	46	218	133	103	49
Total	1,044	613	311	209	1,601	869	520	372

Table 4 *pClust-mr*: Breakdown of runtime (in seconds) by phases for the input graph Pfam-A-11M.

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 5 *pClust-mr*: The run-time (in seconds) for the Shingling Phase I as a function of input and system sizes on the Magellan Hadoop cluster. All runs were performed using $s = 2$ and $c = 100$.

No. map & reduce tasks	*Time (in sec)*		
	Pfam-A-4M	*Pfam-A-8M*	*Pfam-A-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

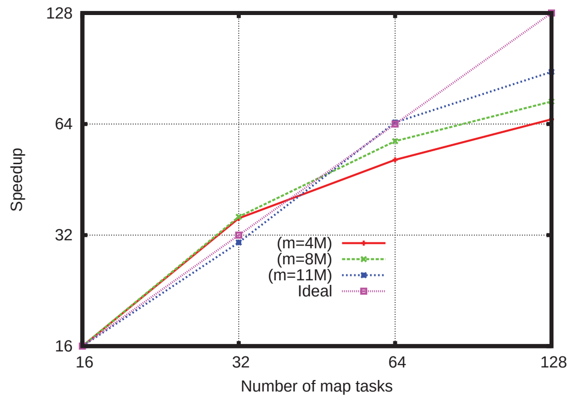

Figure 7 *pClust-mr*: Speedup of the Shingling Phase I.

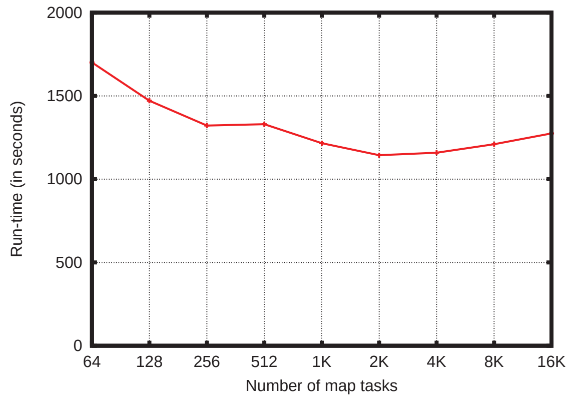

Figure 8 *pClust-mr*: Effect of varying the number of map tasks on Shingling Phase I's runtime for the Pfam-A-11M input. It can be observed that the optimal setting for the number of map tasks is 2K for this input.

cores for the smaller inputs Pfam-A-4M and Pfam-A-8M, and that the linear scaling behavior extends to 64 cores for the larger input (Pfam-A-11M). Figure 7, which shows the relative speedup calculated using the 16 tasks run as the reference, confirms this scaling trend. A peak speedup of ~90x is obtained for the 11M input running using 128 map tasks.

Task granularity study: 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 map tasks from 64 up to 16K. The results are shown in Figure 8 for the Pfam-A-11M graph. 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 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. For this set, we find that the empirical optimal value for task granularity is occurring approximately at 5,500 edges per map task ($\approx \frac{11M}{2K}$).

Reducing space-complexity at the reducers: Each reducer task in *pClust-mr* stores an $O(c \times s)$ matrix to keep track of the running minimum s elements generated for each random trial corresponding to a given source vertex u. This space complexity can be further reduced to $O(s)$. However, observe that there is no interdependency among elements inserted across different random trials. Therefore, an alternative is to dedicate a reducer task to every unique combination of

$< u, j >$, where j is the random trial id. This would reduce the space complexity for each reducer task to $O(s)$. We implemented this alternative version and upon experimenting found that this space improvement comes at the cost of increased runtime — for instance, on the same input and on the same number of cores, reducers took 20 m 6 s using our old configuration, whereas 33 m 5 s using the new modification. The increase in runtime is probably due to an increased overhead in handling more reducer tasks by the Hadoop framework. It is noteworthy that owing to fact that both c and s are relatively small constants (e.g., $c \approx 100$, $s \leq 10$) in practice, retaining the original space complexity of $O(c \times s)$ per reducer task is not prohibitive.

4.2.3 Clustering results

A detailed qualitative analysis of the clustering results is out of scope for this paper as it would entail conducting several rounds of parametric tests and comparisons against benchmarks. Instead, we present here the results of our preliminary analysis on three real world networks — see Table 6. The three inputs represent three different types of networks — CAMERA-100M and Pfam-A-11M are biological graphs but with two different connectivity footprints (higher and lower mean vertex degree, respectively) and graph topology (unipartite and bipartite, respectively); whereas cnr-2000 (12) is an example of a non-biological network, which consists of 325,557 vertices and 2,738,969 edges. We use two measures as an indication of the clustering quality – the average density of clusters and cluster modularity. For modularity calculations, Newman's formula (36) was used for unipartite graphs (CAMERA-100M and cnr-2000) and Murata's formula (32) for bipartite inputs (Pfam-A-11M). Note that the upperbound for the density and modularity would both be 1. It is also to be noted that while modularity is a standard measure to assess community detection algorithms, the Shingling heuristic, which we have implemented for this paper, is a density-driven method although without any density cutoff.

We also compared the results obtained using our shared memory implementation of the Shingling heuristic (*pClust-sm*) against the Louvian method (4). The method is sequential and is one of the most widely used community detection methods. For fairness of comparison, we ran both methods on the same platform, which is the commodity Linux node.

Table 6 shows the results of our preliminary analysis. It can be observed that, with the exception of the clustering modularity for CAMERA-100M, the Louvian tool delivers consistently higher density and modularity values and faster runtimes. More specifically, *pClust-sm* tends to create clusters that are more inclusive and hence less dense than the clusters generated by the Louvian tool. Depending on the nature of input this may or may not be desirable. For instance, while for inputs such as cnr-2000, a higher density (and modularity) indicates stronger cohesiveness, for CAMERA-100M inclusion of peripheral vertices is likely to be more biologically relevant due to the evolutionarily diverse composition in metagenomic communities and low sampling rates. Although these results are presented with a few standard measures such as density and modularity, the appropriateness of a clustering tool to a particular application can be best judged by comparing against curated benchmarks. Without such an evaluation, it is difficult to assess the efficacy of individual clustering methods. That said, based on our observations we posit that for traditional networks where community structure is sought after, a more traditional community detection tool such as the Louvian method could be better suited, whereas, a clustering heuristic such as Shingling could perhaps be better suited for graph inputs where it becomes a challenge to include nodes at the periphery of a cluster to accommodate low sampling rates and data diversity (33; 40). The latter has been corroborated in our earlier studies with the heuristic (43; 42), where the method delivered better results than known clustering results.

5 Conclusion

In this paper, we presented parallel algorithms for an irregular computational problem which is to cluster vertices of a large graph into dense communities. More specifically, we developed two complementary parallel implementations for the Shingling heuristic, one using OpenMP multithreading to take advantage of multicore shared memory platforms, and another using the MapReduce paradigm for distributed memory Hadoop clusters. Studies presented in this paper offer an insight into certain design level challenges and options for implementing a highly irregular graph application. Our shared memory implementation takes advantage of the shared address space and uses multicore parallelization to operate on irregular data structures such as hash tables and union-find. The resulting new algorithm improves both the asymptotic runtime and memory complexities of an older, serial version of the algorithm, and experimental results demonstrate orders of magnitude runtime gains while significantly enhancing the problem size reach.

The MapReduce implementation offers insights on how to carry out these highly irregular operations under a distributed memory setting. Consequently, the resulting algorithm has emerged to be significantly different from the serial algorithm, so as to facilitate extraction of parallelism from the Hadoop MapReduce framework. While the runtimes obtained from the MapReduce implementation is at least an order of magnitude higher than for the shared memory implementation, the linear scaling behavior for the time-dominant phase is a promising indication of the framework's potential to scale to larger systems. The large aggregate memory and the tens of thousands of

Table 6 A summary of clustering results obtained from *pClust-sm* and the Louvian method (4). The commodity Linux node (32 GB RAM, 8 cores) was used to perform all the runs. The results reported for *pClust-sm* correspond to 8 cores, and with input parameters $s = 3$ and $c = 100$. For larger values of s, the density improves only marginally with a more substantial decrease in modularity (e.g., for CAMERA-100M: setting $s = 5$ increases the density to 0.5843 while decreasing the modularity to 0.9179).

Input graph	*Density*		*Modularity*		*Time (in sec)*	
	pClust-sm	Louvian	*pClust-sm*	Louvian	*pClust-sm*	Louvian
CAMERA-100M	0.5540±0.2487	0.7955±0.2319	0.9459	0.9349	372	115
cnr-2000	0.5457±0.3288	0.6134±0.3213	0.6615	0.8129	55	6
Pfam-A-11M	0.9552±0.1211	0.9932±0.0463	0.3858	0.8276	192	39

cores available in modern day large-scale clusters make it an attractive option to resort to while analyzing very large inputs.

Some important improvements have been planned for the near future. The high scalability, low runtimes of our shared memory code coupled with the high quality in clustering have encouraged us to test out very large inputs that could be constructed from current metagenomics sequence repositories such as CAMERA. Also, we plan to combine the runtime and memory benefits in shared memory architectures with the programming simplicity of the MapReduce paradigm using libraries such as shared memory MapReduce libraries such as Phoenix (41).

Acknowledgments

This paper is a revised and expanded version of two workshop papers (9; 39). We sincerely thank the anonymous reviewers of these papers for their constructive feedback. We also thank the support staff at National Energy Research Scientific Computing (NERSC) and at Remote Data Analysis and Visualization Center (RDAV) for their generous help and support while using their systems, the Magellan Hadoop cloud system and the Nautilus supercomputer, respectively. This research was supported in parts by DOE award DE-SC-0006516 and an REU supplement attached to NSF grant IIS 0916463.

References

[1] Andersen, R. and Chellapilla, K. (2009) ‘Finding dense subgraphs with size bounds’, *Lecture Notes in Computer Science*, Vol. 5427, pp.25–36.

[2] Andreopoulos, B., An, A., Wang, X. and Schroeder, M. (2009) ‘A roadmap of clustering algorithms: finding a match for a biomedical application’, *Briefings in Bioinformatics*, Vol. 10, No. 3, pp.297–314.

[3] Bader, G. and Hogue, C. (2003) ‘An automated method for finding molecular complexes in large protein interaction networks’, *BMC Bioinformatics*, Vol. 4, No. 2, pp.1471–2105.

[4] Blondel, V.D., Guillaume, J., Lambiotte., R., and Lefebvre., E. (2008) ‘Fast unfolding of communities in large networks’, *Journal of Statistical Mechanics: Theory and Experiment*, Vol. 2008, pp.P10008.

[5] Broder, A.Z., Charikar, M., Frieze, A., and Mitzenmacher, M. (2000) ‘Min-wise independent permutations’, *Journal of Computer and System Sciences*, Vol. 60, pp.630–659.

[6] Broder, A.Z., Glassman, S., Manasse, M. and Zweig, G. (1997) ‘Syntactic clustering of the web’, *WWW6/Computer Networks*, Vol. 29, pp.1157–1166.

[7] Brohee, S. and Helden, J.V. (2006) ‘Evaluation of clustering algorithms for protein-protein interaction networks’, *BMC Bioinformatics*, Vol. 7, p.488.

[8] ‘CAMERA - Community Cyberinfrastructure for Advanced Microbial Ecology Research & Analysis’, *http://camera.calit2.net* . Last date accessed 6/12/2012.

[9] Chapman, T. and Kalyanaraman, A. (2011) ‘An OpenMP algorithm and implementation for clustering biological graphs’, *Proc. IAAA’11 - Workshop on Irregular Applications: Architectures & Algorithms*, Held in conjunction with SC’11, Seattle, WA, p.3-10.

[10] Charikar, M. (2000) ‘Greedy approximation algorithms for finding dense components in a graph’, *Lecture Notes in Computer Science*, Vol. 1913, pp.139–152.

[11] Clauset, A., Newman. M.E.J., and Moore, C. (2004) ‘Finding community structure in very large networks’, *Phys. Rev. E*, Vol. 70, No. 6, pp.066111.

[12] ‘cnr-2000: A very small crawl of the Italian CNR domain’, *http://law.dsi.unimi.it/webdata/cnr-2000/*. Laboratory for web algorithmics, Last date accessed 6/12/2012.

[13] Dean, J. and Ghemawat, S. (2008) ‘MapReduce: simplified data processing on large clusters’, *Communications of the ACM*, Vol. 51, No. 1, pp.107–113.

[14] D’haeseleer, P. (2005) ‘How does gene expression clustering work?’, *Nat Biotech*, Vol. 23, pp.1499–1501.

[15] Emrich, S., Kalyanaraman, A. and Aluru, S. (2005) ‘Chapter 13: Algorithms for large-scale clustering and assembly of biological sequence data’, *Handbook of computational molecular biology*, CRC Press.

[16] Enright, A.J., Van Dongen, S. and Ouzounis, S.A. (2002) ‘An efficient algorithm for large-scale detection of protein families’, *Nucleic Acids Research*, Vol. 30, No. 7, pp.1575–1584.

[17] Feige, U., Kortsarz, G. and Peleg, D. (2001) ‘The dense k-subgraph problem’, *Algorithmica*, Vol. 29, pp.410–421.

[18] Finn, R., Mistry, J., Tate, J. *et al.* (2010) 'The Pfam protein families database', *Nucleic Acids Research*, Vol. 38, No. 1, pp.D211–D222.

[19] Gibson, D., Kumar, R. and Tomkins, A. (2005) 'Discovering large dense subgraphs in massive graphs', *Proceedings of the International Conference on Very Large Data Bases*, pp.721–732.

[20] Goldberg, A.V. (1984) 'Finding a maximum density subgraph', *Technical Report CSD-84-171*, UC Berkeley.

[21] Jeong, H., Mason, S.P., Barabasi, A. and Oltvai, Z.N. (2001) 'Lethality and centrality in protein networks', *Nature*, Vol. 411, pp.41–42.

[22] Kalyanaraman, A. and Aluru, S. (2005) 'Chapter 12: Expressed Sequence Tags: Clustering and applications', *Handbook of computational molecular biology*, CRC Press.

[23] Kang, U., Tsourakakis, C. E. and Faloutsos, C. (2009) 'PEGASUS: A peta-scale graph mining system – implementation and observations', *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM 2009)*, pp.229–238.

[24] Khuller, S., Saha, B., Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S. and Thomas, W. (2009) 'On Finding Dense Subgraphs', *Automata, Languages and Programming*, Springer Berlin / Heidelberg, pp.597–608.

[25] Lam, H., Deutsch, E.W., Eddes, J.S., Eng, J.K., Stein, S.E. and Aebersold, R. (2008) 'Building consensus spectral libraries for peptide identification in proteomics', *Nat Meth*, Vol. 5, pp.873–875.

[26] Lancichinetti, A. and Fortunato, S. (2009) 'Community detection algorithms: A comparative analysis', *Phys. Rev. E*, Vol. 80, No. 056117.

[27] Lawler, E. (1976) 'Combinatorial optimization - networks and matroids', *New York: Holt, Rinehart and Winston*.

[28] Lee, V.E., Ruan, N., Jin, R. and Aggarwal, C. (2010) 'A Survey of Algorithms for Dense Subgraph Discovery', *Managing and Mining Graph Data*, Springer US, pp.303–336.

[29] Ma, H. and Zeng, A. (2003) 'The connectivity structure, giant strong component and centrality of metabolic networks', *Bioinformatics*, Vol. 19, pp.1423–1430.

[30] Madduri, K., Ediger, D., Jiang, K., Bader, D.A. and Chavarria-Miranda., D.G. (2009) 'A faster parallel algorithm and efficient multithreaded Implementations for evaluating betweenness centrality on massive datasets', *Proceedings of the Workshop on Multithreaded Architectures and Applications*, Rome, Italy.

[31] Matthews, S. and Williams, T. (2010) 'MrsRF: an efficient MapReduce algorithm for analyzing large collections of evolutionary trees', *BMC Bioinformatics*, Vol. 11, No. S15.

[32] Murata, T. (2009) 'Detecting Communities from Bipartite Networks Based on Bipartite Modularities', In *Proc. Computational Science and Engineering*, vol. 4, pp.50–57.

[33] National Research Council (2007) 'The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet', *National Academies Press*, Washington D.C.

[34] Newman. M.E.J. (2010) 'Networks: An introduction (Chapter 11)', *Oxford University Press*.

[35] Newman. M.E.J. (2003) 'The structure and function of complex networks', *SIAM Review*, Vol. 45, pp.167–256.

[36] Newman. M.E.J., and Girvan, M. (2004) 'Finding and evaluating community structure in networks', *Phys. Rev. E*, Vol. 69, No. 2, pp. 026113.

[37] Olman, V., Mao, F., Wu, H. and Xu, Y. (2007) 'A parallel clustering algorithm for very large data sets', *IEEE/ACM Transaction on Computational Biology and Bioinformatics*, Vol. 5, No. 2, pp.344–352.

[38] 'Pfam - Protein family database', `http://pfam.sanger.ac.uk/`. Last date accessed 6/12/2012.

[39] Rytsareva, I. and Kalyanaraman, A. (2011) 'An efficient MapReduce algorithm for parallelizing large-scale graph clustering', *Proc. ParGraph'11 - Workshop on Parallel Algorithms and Software for Analysis of Massive Graphs*, Held in conjunction with HiPC'11, Bengaluru, India.

[40] Simon, C. and Daniel, R. (2011) 'Metagenomic Analyses: Past and Future Trends', *Applied and Environmental Microbiology*, vol. 77, no. 4, pp.1153–1161.

[41] Talbot, J., Yoo, R.M., and Kozyrakis, C. (2011) 'Phoenix++: Modular MapReduce for Shared-Memory Systems', *Proc. International Workshop on MapReduce and its Applications (MAPREDUCE)*, San Jose, CA.

[42] Wu, C. (2011) 'Parallel algorithms for large-scale computational metagenomics', *Ph.D. dissertation*, Washington State University, Pullman, WA, USA. `https://research.wsulibs.wsu.edu/xmlui/handle/2376/2889`. Last date accessed 6/12/2012.

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

[44] Wu, C., Kalyanaraman, A., and Cannon, W.R. (2012) '*pGraph*: Efficient parallel construction of large-scale protein sequence homology graphs', *IEEE Transactions on Parallel and Distributed Systems*, Preprint, DOI *10.1109/TPDS.2012.19*.

[45] Yoo, A., Chow, E., Henderson, K. *et al.* (2005) 'A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L', *Proceedings ACM/IEEE conference on Supercomputing*, pp.25–35.

[46] Yooseph, S., Sutton, G., Rusch, D.B. *et al.* (2007) 'The Sorcerer II Global Ocean Sampling expedition: expanding the universe of protein families', *PLoS Biology*, Vol. 5, No. 3, pp.432–466.

Note

[1]Note that if the input is not a bipartite graph but a 1-mode graph $G' = (V', E')$, then it can be trivially converted into an equivalent bipartite graph $G = (V_l, V_r, E)$ by setting $V_l = V_r = V'$ and drawing edges from E' between the two vertex partitions.

[2]During experimentation, we tested an alternative design where there is one hash table for every fixed number of threads. However, the implementation did not change the

behavior of the code and hence we did not pursue that direction.

[3]While the mappers varied in their respective runtimes, we did *not* identify any compelling evidence for a nonuniform workload distribution. The mean completion time for an average mapper was more than 50 minutes.