An OpenMP Algorithm and Implementation for Clustering Biological Graphs

Timothy Chapman
Jack Baskin School of Engineering
University of California
Santa Cruz, CA, USA
rsog412@gmail.com

Ananth Kalyanaraman
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA, USA
ananth@eecs.wsu.edu

ABSTRACT

Graph algorithms on parallel architectures present an interesting case study for irregular applications. Among the graph algorithms popular in scientific computing, graph clustering or community detection has numerous applications in computational biology. However, this operation also poses serious computational challenges because of irregular memory access patterns, large memory requirements, and their dependence on other auxiliary (also irregular) data structures to supplement processing. In this paper, we address the problem of graph clustering on shared memory machines. We present a new OpenMP-based parallel algorithm called *pClust-sm*, which uses adjacency lists, hash tables and union-find data structures in parallel. The algorithm improves both the asymptotic runtime and memory complexities of a previous serial implementation. Preliminary results show that this algorithm can scale up to 8 threads (cores) of a shared memory machine on a real world metagenomics input graph with 1.2M vertices and 100M edges. More importantly, the new implementation drastically reduces the time to solution from the order of several hours to just over 4 minutes, and in addition, it enhances the problem size reach by at least one order of magnitude.

Keywords

Graph clustering; shared memory parallel algorithm; hash tables; union-find data structure; parallelization techniques and data structures.

1. INTRODUCTION

Biological data, both naturally occurring and synthetically generated, lend themselves well to graph-based representations, where vertices can be used to represent the data points and edges (weighted or unweighted, directed or undirected) can be used to represent the relationship shared between data. Consequently, graph-based representations are a popular way to model problems in computational biology.

IAAA'11, November 13, 2011, Seattle, Washington, USA.

Once modeled as a graph, various scientifically interesting questions can be posed on the data and they typically translate into performing some kind of graph operations — e.g., performing an Euler tour or Hamiltonian path for genome assembly, finding hubs and critical paths in gene regulatory networks, finding connected components to group expressed sequences (transcriptomics), and clustering, which forms the focal point for this paper.

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

Clustering has a number of applications in computational biology. For instance: it can be used to reduce redundancy within sequence repositories; identify complexes within metabolic networks [2]; identify core groups of proteins that constitute a protein family [26, 28, 29] and in the process also help assign family memberships for newly found peptide candidates [29]; help in the construction of mass spectral libraries for peptides [17]; and can be used to condense the space of plausible computer-generated phylogenetic trees [23].

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 [29]. 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, 11, 16] or with large degree polynomial methods [25, 26]. 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 this paper, we parallelize the *Shingling* heuristic devel-

oped by Gibson *et al.* [12]. In our earlier work, we implemented a serial version of this heuristic, and applied it in the context of metagenomic protein family detection [28]. Put briefly, this approach [28], 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 [28, 27] 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 approach by which the large graph problem is first broken, in parallel, 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 to 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. Resorting to secondary storage during processing is one option.

In this paper, we present a shared memory, multi-threaded algorithm and implementation for *pClust*. This new implementation, which we will refer to as *"pClust-sm"*, uses OpenMP for parallelization within a symmetric multiprocessor node. It reduces both the runtime and memory requirements of the previous implementation and enables shared memory parallelization. We replace a sorting step using hash tables, and then use an on-the-fly scheme for reporting clusters using union-find data structure, which reduces the peak memory usage by a big constant factor (≈ 100 in practice). Preliminary results show that *pClust-sm* scales appreciably up to 8 cores for larger inputs on a single node of an SGI Altix shared memory machine. More importantly, it has allowed us to directly solve a real world input 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 [27].

The paper is organized as follows: Section 2 presents a brief overview of the related literature on clustering and dense subgraph detection. In Section 3, we first describe the sequential clustering algorithm and then present our OpenMP parallelization and algorithmic improvements. Experimental results are presented in Section 4, and Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK

The problem of finding a densest subgraph within an input graph is solvable in polynomial time [7, 13, 19]. 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, 11, 16]. 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 [11, 20].

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 [8, 9, 15]). 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. From a community detection standpoint, such methods tend to create loose clusters and suffer from error propagation during incremental construction. A different class of applications benefit from graph clustering based formulations that aim to detect tight communities within biological data (e.g., [2, 10, 14, 21, 29]).

Independently, in other areas of computing such as social and cyber networks, numerous algorithms have been developed for community detection (reviewed in [18, 20, 24]). M.E.J. Newman, in his pioneering work on discovering community structure from networks [25], 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 [25], 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 [22] for efficiently calculating betweenness centrality on graphs. A different approach [26] works on weighted graphs, where edge weights are distance measures, and uses Minimum Spanning Trees (MST) for clustering by taking advantage of the property that closely related groups tend to map to subtrees within an MST. However, this method can also be time consuming ($\Omega(n^2)$) and the method has not been compared with other methods making it difficult to assess its quality.

Gibson *et al.* developed the Shingling approach for identifying web communities [12]. The underlying method (described in Section 3.1) uses a bipartite graph formulation along with a random sampling procedure and secondary sorting to determine dense subgraphs. In [28], 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 im-

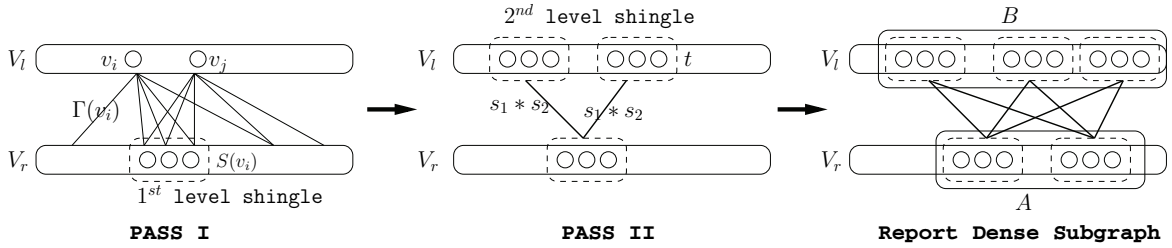

Figure 1: Illustration of the two-pass Shingling algorithm. Typically, $s_1 = s_2 = s$.

plementation proposed in this paper reduces this runtime by using a hash table and through parallelization under the OpenMP model; it also reduces the memory complexity to $O(n \times c)$.

3. METHODS

3.1 The serial algorithm and *pClust* implementation

Let $G(V, E)$ be the input (undirected, unweighted) graph with n vertices and m edges. Let $B = (V_l, V_r, E)$ be the bipartite obtained from $G(V, E)$ by setting $V_l = V_r = V$ and preserving all edges of G between the two partitions. We note here that it suffices to implement B using an adjacency list representation for V_l (or equivalently V_r), and therefore it can be guaranteed that the space required to store B is no more than the space required to store G. Given a vertex $u \in V_l \bigcup V_r$, let $\Gamma(u)$ denotes the set of its out-links — i.e., $\Gamma(u) = \{v \mid (u, v) \in E\}$.

DEFINITION 1. *Given parameters (s, c), the term "shingle" [4] of a vertex u is used to denote an arbitrary $s-$element subset of $\Gamma(u)$, and the term "$< s, c >$shingle set of u" is used to denote a set of c shingles of u, each of size s.*

The main idea of the Shingling algorithm is as follows: Intuitively, two vertices sharing a shingle, by definition, share s of their out-links. The algorithm seeks to group such vertices together and use them as building blocks for dense subgraphs. Larger the value of s, lesser the probability that two vertices share a shingle. The parameter c is intended to create the opposite effect. In addition, this parameter offers a way to restrict the computation space for detecting similar pairs of vertices, as it is not computationally practical to exhaustively compute the intersection of shingles generaed for every pair of vertices. This is achieved by using the min-wise independent permutation property [3]. Instead of generating c arbitrary shingles for a vertex v, the algorithm first generates c randomly sorted permutations of $\Gamma(v)$ and selects the s minimum elements from each permutation. Even if two vertices share a modest number of out-links, the randomness in this property will ensure that the probability of the vertices sharing a shingle becomes sufficiently high. In other words, the parameter c represents the number of random trials. This will be particularly helpful for detecting large subgraphs, as they are expected to be less dense

The algorithm is implemented in *pClust* in three phases (as illustrated in Figure 1):

- **Shingling Phase I:** An $< s, c >$shingle set (denoted by $S(v_i)$) is generated for each vertex $v_i \in V_l$. For ease of implementation, each shingle is mapped to an integer using a hash function. The results are recorded as a 2-tuple $<s(v_i), v_i>$, where $s(v_i) \in S(v_i)$. Next, vertices sharing the same shingle are grouped. This is achieved by sorting the tuples based on shingle values. The resulting tuple list is input to the second phase. Let $S_1 = \bigcup_{i=0}^{n} S(v_i)$ denote the set of shingles generated in this phase; each shingle is referred to as a *first level shingle*.

- **Shingling Phase II:** The algorithm reverses direction and generates an $< s, c >$shingle set for each first level shingle $s(v_i)$, treating each shingle as a source vertex and the vertices $u \in V_l$ that generated it as its neighbors. The result is a set of *second level shingles* S_2, constituted by vertices in V_l.

- **Phase III:** In the final reporting step, all connected components, defined by first level to second level shingle connections, are enumerated and their constituent vertices recorded. This step uses the union-find data structure to perform the union of vertices. Consequently, the set of clusters defined by the union-find data structure 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 $O(|S_1| \times c \times s^2)$, 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 $O(max\{|S_1|, |S_2|\})$, and is $\Theta(n \times c^2)$ in the worst-case.

3.2 pClust-sm: 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[1]. 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_1^id ← null;
Init: Hash Table H with n × c entries;
#pragma omp parallel default(shared)
for (i = 0; i < n; i++) do
  Let u_i = i^th vertex in V_l;
  for (j = 0; j < c; j++) do
    s ← Generate j^th shingle from Γ(u_i);
    Insert(< s, u_i >) into H;
    if first time an entry for s is inserted in H then
      Append inserted location for s to S_1^id;
    end if
  end for
end for
Init: UnionFind UF[1 ... n];
#pragma omp parallel default(shared)
Init: k = 0;
repeat
  Let next ← S_i^id[k++];
  Let s ← shingle object pointed to by next;
  for (j = 0; j < c; j++) do
    t(s) ← Generate j^th shingle from Γ(s)
    Update UF ← {⋃_{∀v_i∈s} v_i} ⋃ {⋃_{∀u_i∈t(s)} u_i}
  end for
until next == null
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

[1]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.

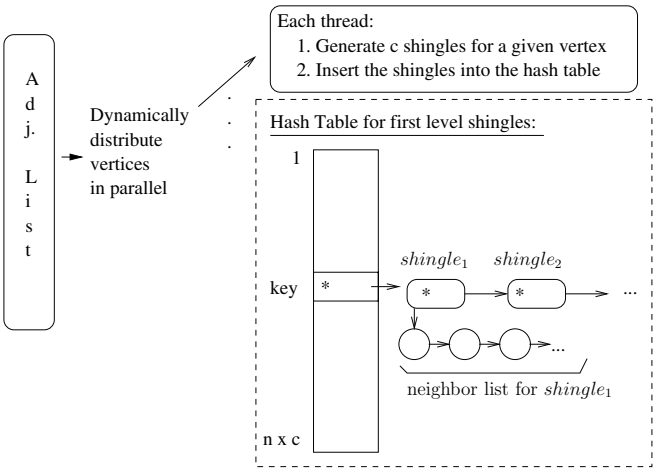

Figure 2: Illustration of Phase I and the hash table shared by all threads in *pClust-sm*. Asterisks indicate the resolution at which locks could be potentially placed while inserting.

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 ← hash("v1#v2# ... #v_s'') % tablesize;
Let fetch ← 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 ← 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 our new implementation *pClust-sm*, we completely elim-

inate 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 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 executed 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). However, our experimental results, at least for the inputs tested, did not result in such an imbalance. That said, we plan to implement and evaluate the other scheme in the near future.

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 reattempting until no other threads have updated the entry. In our experiments, we never encountered re-attempts.

3.3 Analysis

Phase I: In the current implementation, we initialize the hash table with size equal to $n \times c$ as that is the upperbound on the number of distinct first level shingles. This makes the memory complexity for Phase I $O(n \times c)$. As for the runtime complexity, the worst-case runtime for Phase I is $O(\frac{n \times c \times s^2}{p})$, where p is the number of OpenMP threads (assuming each thread is assigned one processing core). Note the conspicuous absense of the sorting time. In practice, the runtime would also depend on vertex degree distribution.

Combined Phases II and III: The worst-case runtime for generating second level shingles is $O(\frac{|S_1| \times c \times s^2}{p})$, where $c \leq |S_1| \leq n \times c$ depending on the input. 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.

Algorithm 3 Union (Input: a, b)

```
done ← false;
repeat
  Let root_a ← Find(a);
  Let root_b ← 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 ← true;
    end if
  end if
until done
```

3.4 Implementation

The program *pClust-sm* was written in C++ and OpenMP. In the current implementation, the constant of proportionality in the memory complexity $O(n \times c)$ 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/`.

4. EXPERIMENTAL RESULTS

4.1 Experimental setup

We used the NICS *Nautilus* supercomputer as our experimental platform. Nautilus 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 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.

As for the input, we used the same metagenomic sequence sets used in our earlier experiments using the serial version *pClust* [28] so that the answers can be matched for correctness and performance improvements of *pClust-sm* over *pClust* can be measured. Thar largest of this input contains 1,280,000 vertices (metagenomic peptide sequences download from the CAMERA website [6]) and 100,919,106 edges connecting them. Smaller subgraphs were extracted from this large graph containing approximately 50M, 30M and 20M edges.

4.2 Performance results

Table 1 shows the runtime of *pClust-sm* as a function of the input size. The loading phase is currently a serial component in the code 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;

Input graph size	Loading phase	Remaining phases				
		Number of threads (p)				
		1	2	4	8	16
$(n = 1.2M, m = 100M)$	40	1044	613	311	209	195
$(n = 325K, m = 50M)$	15	474	241	155	108	94
$(n = 190K, m = 30M)$	10	282	156	112	82	60
$(n = 189K, m = 20M)$	8	266	159	105	69	69

Table 1: The run-time (in seconds) of *pClust-sm* on various input and system sizes. n denotes the number of vertices and m denotes the number of edges. All runs were performed using $s = 3$ and $c = 100$.

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 1 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 set (n=1.2M, m=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 3 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. For larger input sizes, it will become important to scale beyond 16 threads and therefore more work is needed on this front.

The results 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 previous serial version of the *pClust* code. The latter approach is an indirect approach in which the graph was split into 65K connected components and then our serial code was run on each of the connected component individually (as it represents an independent subproblem). The time taken to cluster all 65K components in parallel on a 128 node cluster was ∼30 minutes [27]. Put another way, our new shared memory implementation, *pClust-sm*, is an approach that operates on the input graph directly and reduces the time to solution for 1.2M vertices input from 64 CPU hours to 32 CPU minutes. Note that, in principle, the same technique of breaking the input graph into connected components and then processing each connected component independently can also be used in tandem with *pClust-sm*.

The above comparison shows the effectiveness of the algorithmic improvements (in addition to parallelization) proposed in this paper — viz. replacement of the sorting step with hash table; on-the-fly generation and merging of second level shingles with first level shingles; and eliminating the overhead associated with processing tens of thousands connected components individually (and instead process the entire graph directly as one input).

As further investigation of performance, we recorded the phase-wise breakdown of the total runtime in *pClust-sm*. Table 2 shows the results for the 1.2M data set. Given that the loading phase is a serial step, it does not change with the number of threads. As can be expected, Phase I is the

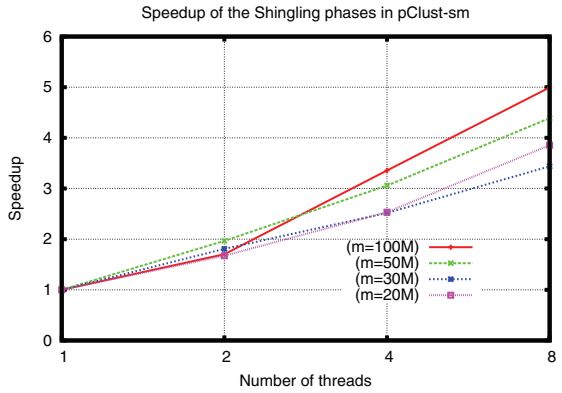

Figure 3: Speedup of *pClust-sm* up to 8 threads.

Phase	Number of threads (p)			
	1	2	4	8
Loading	40	40	40	40
Phase I	780	467	236	163
Combined Phases II & III	264	146	75	46
Total	1084	653	351	249

Table 2: Breakdown of runtime (in seconds) by the different phases of the *pClust-sm* algorithm for the input graph ($n = 1.2M$, $m = 100M$).

dominant phase, and it scales linearly up to 4 threads after which the scaling deteriorates at 8 threads (236 s to 163 s). After incorporating more timing information, 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 ≈50M insertions). These observations leave open the possibility that the slowdown is caused by the increased rate of memory access generated by the increased number of threads.

The runtime for Phases II and III, which execute in tandem, demonstrates better scaling than Phase I up to 8 threads, and demonstrates the effectiveness of parallelizing the union-find data structure.

5. CONCLUSIONS

We presented a new shared memory, OpenMP parallel algorithm called *pClust-sm* for graph clustering and applied it to a real world protein sequence homology graph consisting of 1.2M vertices and 100M edges. The new algorithm improves both the asymptotic runtime and memory worst-case complexities of an older, serial version of the algorithm using algorithmic heuristics and parallelization. Experimental results carried out using this preliminary version of our implementation have demonstrated substantail runtime gains while enhancing the problem reach at least by an order of magnitude.

Although tested only on biological graphs, we expect our algorithm and techniques to extend to other domains as well where bipartite graph clustering can be of use. Furthermore, as described in the original *pClust* paper [28], the reach of the method can be extended to standard graph clustering as well.

Studies presented in this paper offer an insight into certain design level challenges and options for implementing data structures such as hash tables and union-find on shared memory machines. However, the implementation reported here is by no means a perfect solution and has left us open with several possible improvements and extensions. As future work, we plan to work on a few more algorithmic improvements such as: parallelizing the loading phase; and introduce dynamic distribution of first level shingles during Phase II (for second level shingle generation). More importantly, our goals are to extend the linear scaling trend beyond 16 cores, and conduct larger scale experiments that involve graphs with possibly tens to hundreds of millions of vertices. Another important step is to investigate the extension of these techniques to other shared memory architectures such as the Cray XMT.

6. ACKNOWLEDGMENTS

The research was supported by an REU supplement attached to the NSF grant IIS-0916463.

7. REFERENCES

[1] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. *Lecture Notes in Computer Science*, 5427:25–36, 2009.

[2] G. Bader and C. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4:2, 2003.

[3] A.Z. Broder, M. Charikar, A. Frieze and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:630–659, 2000.

[4] A.Z. Broder, S. Glassman, M. Manasse and G. Zweig. Syntactic clustering of the web. *WWW6/Computer Networks*, 29:1157–1166, 1997.

[5] S. Brohee and J.V. Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC Bioinformatics*, 488:488, 2006.

[6] CAMERA - Community Cyberinfrastructure for Advanced Microbial Ecology Research & Analysis. `http://camera.calit2.net`. Last date accessed (9/15/2011).

[7] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. *Lecture Notes in Computer Science*, 1913:139–152, 2000.

[8] P. D'haeseleer. How does gene expression clustering work? *Nat Biotech*, 23:1499–1501, 2005.

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

[10] A.J. Enright, S. Van Dongen, and S.A. Ouzounis. An efficient algorithm for large-Scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.

[11] U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. *Algorithmica*, 29:410–421, 2001.

[12] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. International Conference on Very Large Data Bases*, pp. 721–732, 2005.

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

[14] H. Jeong, S.P. Mason, A. Barabasi, and Z.N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41–42, 2001.

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

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

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

[18] A. Lancichinetti and S. Fortunato.. Community detection algorithms: A comparative analysis. *Phys. Rev. E*, 80:056117, 2009.

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

[20] V.E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A Survey of Algorithms for Dense Subgraph Discovery. *Managing and Mining Graph Data*, A.K. Elmagarmid, C.C. Aggarwal, and H. Wang, Eds., Springer US, pp. 303–336, 2010.

[21] H. Ma and A. Zeng. The connectivity structure, giant strong component and centrality of metabolic networks. *Bioinformatics*, 19:1423–1430, 2003.

[22] K. Madduri, D. Ediger, K. Jiang, D.A. Bader and D.G. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded Implementations for evaluating betweenness centrality on massive datasets. *Proc. Workshop on Multithreaded Architectures and Applications*, Rome, Italy, May 29, 2009.

[23] S. Matthews and T. Williams. MrsRF: an efficient MapReduce algorithm for analyzing large collections of evolutionary trees. *BMC Bioinformatics*, 11:S15, 2010.

[24] M.E.J. Newman. Networks: An introduction. *Oxford University Press*, 2010.

[25] M.E.J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[26] V. Olman, F. Mao, H. Wu, and Y. Xu. A parallel clustering algorithm for very large data sets. *IEEE/ACM Transaction on Computational Biology and Bioinformatics*, 5(2):344–352, 2007.

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

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

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