Parallel and Scalable Precise Clustering for Homologous Protein Discovery

Stuart Byma
stuart.byma@epfl.ch
EPFL

Akash Dhasade
akashdhasade@gmail.com
IIT

Adrian Altenhoff
adriaal@inf.ethz.ch
ETH Zürich

Christophe Dessimoz
christophe.dessimoz@unil.ch
University of Lausanne

James R. Larus
james.larus@epfl.ch
EPFL

Abstract

This paper presents a new, parallel implementation of clustering and demonstrates its utility in greatly speeding up the process of identifying homologous proteins. Clustering is a technique to reduce the number of comparison needed to find similar pairs in a set of n elements such as protein sequences. *Precise clustering* ensures that each pair of similar elements appears together in at least one cluster, so that similarities can be identified by all-to-all comparison in each cluster rather than on the full set. This paper introduces ClusterMerge, a new algorithm for precise clustering that uses transitive relationships among the elements to enable parallel and scalable implementations of this approach.

We apply ClusterMerge to the important problem of finding similar amino acid sequences in a collection of proteins. ClusterMerge identifies 99.8% of similar pairs found by a full $O(n^2)$ comparison, with only half as many operations. More importantly, ClusterMerge is highly amenable to parallel and distributed computation. Our implementation achieves a speedup of 604× on 768 cores (1400× faster than a comparable single-threaded clustering implementation), a strong scaling efficiency of 90%, and a weak scaling efficiency of nearly 100%.

1 Introduction

The ongoing revolution in genome sequencing is generating large and growing datasets whose value is exposed through extensive computation. The cost of this analysis is an impediment to analyzing these databases when the time for processing grows rapidly as a dataset becomes larger, more inclusive, and more valuable. Asymptotically efficient algorithms are desirable, but sometimes a tradeoff between speed and precision requires the use of expensive algorithms. In this situation, the time to perform an analysis can be reduced by running on a parallel computer or cluster of computers. This paper describes a new approach to applying parallel computing to protein clustering, an important technique in the field of *proteomes*, the analysis of the protein sequences contained in an organism's genome.

Similarities among protein sequences are used as proxies to infer common ancestry among genes. Similar genes are referred to as *homologs*, and their detection allows the transference of knowledge from well-studied genes to newly sequenced ones. Homologs, despite having accumulated substantial differences during evolution, often continue to perform the same biological function. In fact, most of today's molecular-level biological knowledge comes from the study of a handful of model organisms, which is then extrapolated to other life forms, primarily through homology detection. Several sequence homology techniques are among the 100 most-cited scientific papers of all time [24].

Current approaches to find similar (homologous) proteins are computationally expensive. The baseline is to perform an exhaustive, all-against-all ($O(n^2)$) comparison of each sequence against all others using the Smith-Waterman (S-W) or another, similarly expensive ($O(n^2)$) string matching algorithm. This naive approach finds all similar pairs, but it scales poorly as the number of proteins grows. Several databases of similar proteins produced by this approach exist, including OMA [2] and OrthoDB [26]. Analyzing their contents is costly. OMA for example has consumed over 10 *million* CPU hours, but includes proteins from only 2000 genomes.

The large amount of data produced by many laboratories requires new methods for homology detection. In a report published in 2014, the *Quest for Orthologs* consortium, a collaboration of the main cross-species homology databases, reported: "*[C]omputing orthologs between all complete proteomes has recently gone from typically a matter of CPU weeks to hundreds of CPU years, and new, faster algorithms and methods are called for*" [21]. Ideally, a new algorithm with asymptotically better performance would find the same similarities as the ground truth, all-against-all comparison. Unfortunately, fast (sub $O(n^2)$) algorithms — based on k-mer counting, sequence identity, or MinHash — identify significantly fewer homologs and hence are not practical for this application. In the absence of a better algorithm, a scalable parallel implementation of an $O(n^2)$ solution would help keep pace with the production of sequence data.

Our approach extends the idea of clustering [27] into *precise clustering*, which ensures that each pair of similar proteins appears together in at least one cluster[1]. Similar pairs are then easily identified in the resulting clusters. Traditional clustering techniques such as k-means, hierarchical clustering, density/spatial clustering, etc. are difficult to apply because they partition, require a similarity matrix, or generally do not achieve sufficient selectivity. Our technique uses the transitivity of similarity to construct clusters and to avoid unnecessary comparisons. The key idea is that some *similar* sequence pairs will have the stronger property of *transitively similarity*. Formally, if A is transitively similar to B, and B is similar to C, then C will be similar to A. This not only finds similarity between sequences A and B, but also between A and all sequences similar to B. Transitivity avoids a large number of comparisons and reduces the computational cost.

We exploit transitivity by building clusters of sequences centered on a *representative* sequence to which all cluster members are similar. Any sequence *transitively similar* to a cluster representative is added to its cluster. It need not be compared against the other cluster members, as transitivity implies its similarity with them. Sequences that are only *similar* to the representative are also added to the cluster, but they must also be made representatives of their own, new cluster to ensure they are available for subsequent comparisons. In this way, all similar pairs end up together in at least one cluster. Previous work showed that this approach performs well for protein clustering [27], but the greedy implementation in that paper was slow and not scalable.

In this paper, we generalize the problem of clustering with a transitive relation, introduce a parallel and distributed algorithm, and apply our approach to clustering protein sequences. Our new algorithm for precise clustering is called *ClusterMerge*. The key insight enabling parallelism is that two clusters can be *merged* if their representatives are transitively similar since each cluster's members are similar to the other cluster's representative (and members). Members of both clusters can be merged into a single cluster with one representative. If the representatives are similar (but not transitively similar), the clusters exchange elements that are similar to the other cluster's representatives. The result of merging is either one cluster or two *unmergeable* clusters (since their representatives are not transitively similar). Merging clusters reframes clustering as a process that starts with single-element clusters containing each element in the dataset and merges them bottom-up until a set of unmergeable clusters remains.

ClusterMerge exposes a large amount of parallelism in its tree-like computation. However, the computation is highly irregular because of the wide span in the length of proteins (hundreds to tens of thousands of amino acids), the $O(n^2)$ string comparison that exaggerates this disparity, and differences in the size of clusters, all of which requires dynamic load balancing to achieve good performance. We present efficient parallel and distributed implementations using this cluster merge approach. Our single-node, shared-memory design scales nearly linearly and achieves a speedup of 21× on a 24 core machine. Our distributed design achieves a speedup of 604× while maintaining a strong scaling efficiency of 79% on a distributed cluster of 768 cores (90% on larger datasets), running 1400× faster than the incremental greedy clustering of Wittwer et al. [27]. Our distributed implementation exhibits a weak scaling efficiency of nearly 100% on 768 cores. ClusterMerge and our implementations for protein sequence clustering are open-sourced [25]

This paper makes the following contributions:

- A formalization of precise clustering using similarity and transitivity.
- An algorithm, ClusterMerge, that reformulates the clustering process in a parallelism-friendly form.
- An application of ClusterMerge to the problem of clustering protein sequences that maintains near-perfect accuracy while achieving high parallel efficiency.

The rest of this paper is organized as follows: §2 reviews related work in clustering and sequence clustering. §3 formalizes precise clustering and presents the ClusterMerge algorithm. §4 shows how to apply ClusterMerge to precise protein sequence clustering. §5 discusses our shared memory and distributed implementations of ClusterMerge. §6 evaluates the algorithm, systems, and their performance in this application. §7 discusses future work and §8 concludes.

2 Related Work

Clustering in general has been the subject of considerable research. Andreopoulos et al. survey uses of the techniques in bioinformatics [4]. Widely known techniques are difficult to apply to protein clustering, however.

Partitioning algorithms require an equivalence relation between elements, which is stronger than the not-necessarily transitive similarity relationship in protein clustering. k-means clustering requires a target number of clusters, which is unknown in advance for proteins, and partitions the set. Hierarchical methods partition elements into a tree and preserve hierarchy among elements, but generally require a similarity matrix to exist, which is not the case for our problem, and are expensive ($O(n^3)$). Of particular note is *agglomerative* hierarchical clustering, which also uses bottom-up merge, e.g., ROCK [9]. Density-based clustering uses a local density criterion to locate subspaces in which elements are dense; however, they can miss elements in sparse regions and generally cannot guarantee a precise clustering. Density-based

[1]Since a protein sequence can be in more than one cluster, clustering is not partitioning.

techniques have received attention from the parallel computing community, with the DBSCAN [19] and OPTICS [5] algorithms being parallelized by Patwary et al. [17, 18]

An additional complication of these methods is that they rely on distance metrics in normed spaces, e.g., Euclidean distance, which are usually cheap to compute. Edit distance however, is not a norm and is expensive to compute. Although pure edit distance (i.e., Levenshtein distance) can be embedded in a normed space [15], it is not clear if the gapped *alignment* necessary for protein similarity can be as well.

Clustering of biological sequences is the subject of considerable research. Many of these clustering algorithms employ iterative greedy approaches that construct clusters around representative sequences, a sequence at a time. If the sequence is similar to a cluster representative, it is placed in that cluster. If the sequence is not similar to any existing cluster representative, a new cluster is created with the input sequence as its representative. Some approaches use k-mer counting to approximate similarity (CD-HIT [12], kClust [10], Mash [14]), while others use sequence identity, i.e., the number of exact matching characters (UCLUST [7]). Of note is Linclust [22], an approach that operates in linear time by selecting m k-mers from each sequence and grouping sequences that share a k-mer. The longest sequence in a group is designated its *center* and other sequences are compared against it, avoiding a great deal of computation.

Unfortunately, sequence identity and k-mers are unsuitable for finding many homologs. Protein alignment substitution matrices are heterogeneous (e.g., BLOSUM62 [11]) since distinct amino acids may be closely related. Hence, protein sequences that appear different — with low sequence identity and therefore few or no shared k-mers — can often have high alignment scores. These similar pairs will be missed by k-mer-based clustering techniques. For example, the fraction of similar sequence pairs found by kClust, UCLUST, MMSeqs2 linclust, and MMSeqs2 are 10.4%, 13.5%, 0.5%, and 36.4%, respectively.

Wittwer et al. [27] use an iterative greedy approach to cluster protein sequences, using transitivity in the data to avoid comparing each sequence with all others while recovering ~99.9% of similar pairs. Our work uses a similar transitivity function. However, the previous iterative greedy approach is slow and difficult to parallelize because each added sequence depends on the clusters from the previous sequences and requires fine-grained synchronization.

3 Precise Clustering

S is a set of elements. Elements in S can be compared using a similarity function $f(i, j)$ that returns a measure of the similarity between elements i and j. We wish to find all element pairs (i, j) in S such that $f(i, j) > T$, where T is a threshold parameter. We call these pairs *significant pairs*.

While significant pairs can be found by pairwise of comparison of all elements in S, this requires $O(n^2)$ comparisons. To avoid examining *all* possible pairs, we cluster elements in S such that for all element pairs (i, j) in S where $f(i, j) > T$, both i and j are members of at least one cluster. We call this a *precise clustering*. A cluster C is a subset of S defined as follows:

$$\forall e \in C, f(e, r_C) > T \tag{1}$$

where r_C is the unique *representative* element of the cluster.

The similarity function f is not an equivalence function — elements in a cluster are similar to its representative, but not necessarily to each other (although that may be likely). In addition, a representative is not required to be similar to other elements similar to its cluster members. To identify all significant pairs in S, each element would need its own cluster, and the problem devolves to all-against-all comparison.

Therefore, to avoid this, we exploit a stronger property of *transitive similarity*. Formally, for elements i, j and k, if i is transitively similar to j, and i is similar to k, then j is similar to k. Therefore, if i is the representative of a cluster, then we can infer similarity between j and every other element in the cluster. We require an additional *transitivity function* $R(i, j)$ defined:

$$\forall (i, j, k) \in S, R(i, j) \implies f(i, k) > T \wedge f(j, k) > T$$

$R(i, j)$ tells us that elements i and j are *transitively similar*. Element j can be clustered with element i and its similarity with other cluster members (k) inferred. This reduces the number of comparisons needed to form a precise clustering.

3.1 Merging Clusters

Our key to exposing parallelism lies in recognizing that clusters with transitively similar representatives can be *merged*. This allows us to reframe clustering as a series of *cluster merges*. Two clusters can be merged as follows. First, the representatives are compared using the similarity function f. If they are similar, the transitivity function R is applied to see if they are transitively similar. If so, the clusters can be combined into a single cluster, with one representative for all elements. Otherwise, if the representatives are only similar but not transitive, members of either cluster *might* be similar to the other representative. To avoid missing these significant pairs, each cluster is compared against the other's representative and the similar elements are duplicated in the other cluster. Finally, if the representatives are not similar, both clusters remain unchanged. The result is a set of one or two clusters that are no longer mergeable.

Merging can also be applied to two *sets* of clusters. Algorithm 1 describes the process in detail. Each cluster in the first set (*cs1*) is compared to and possibly merged with every cluster in the second set (*cs2*). For each cluster pair, the process described above is applied. Finally, all unmergeable clusters are returned in a new set.

Algorithm 1 Cluster Set Merge

```
procedure MERGE(cs1, cs2)                    ▷ merge cluster set 1, 2
    newClusterSet ← ∅
    for cluster1 in cs1 do
        for cluster2 in cs2 do
            if (cluster2.HasBeenMerged) continue
            s ← f(cluster1.rep, cluster2.rep)  ▷ Similarity
            if (s < T) continue
            if cluster2.IsTransitive(cluster1) then
                cluster2.Objs.append(cluster1.Objs)
                cluster1.HasBeenMerged ← True
                break
            else if cluster1.IsTransitive(cluster2) then
                cluster1.Objs.append(cluster2.Objs)
                cluster2.HasBeenMerged ← True
            else
                ExchangeSimilar(cluster1, cluster2)
            end if
        end for
        if !cluster1.IsMerged then
            newClusterSet.append(cluster1)
        end if
    end for
    for cluster2 in cs2 do                   ▷ add unmerged clusters
        if !cluster2.HasBeenMerged then
            newClusterSet.append(cluster2)
        end if
    end for
    return newClusterSet
end procedure
```

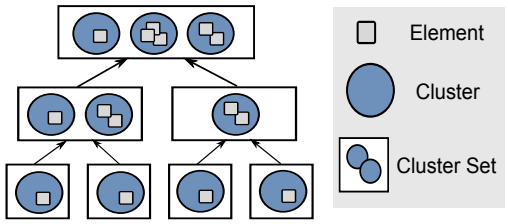


Figure 1. ClusterMerge algorithm. Elements are placed in trivial clusters which are then merged until an unmergeable set remains.

3.2 ClusterMerge Algorithm

The ClusterMerge algorithm uses cluster merging to perform precise clustering. Each element is initially placed in its own cluster as its representative and each cluster is placed in its own set. Algorithm 1 is then applied to merge cluster sets in a bottom-up fashion as depicted in Figure 1.

Algorithm 2 describes this bottom-up merge process. To start, a new cluster set is created for each element, with a single cluster containing only that element. These cluster sets are added to a FIFO queue of sets to merge (the *setsToMerge* queue). The algorithm pops two sets off the queue, merges them using Algorithm 1, and pushes the resulting cluster set onto the queue. The process terminates when only one set is left. This algorithm forms the basis of our implementations further described in §5.

Algorithm 2 ClusterMerge

```
procedure BOTTOMUPMERGE(elements)
    setsToMerge ← Queue()
    for e in elements do
        setsToMerge.push(new ClusterSet(e))
    end for
    while setsToMerge.size() > 1 do
        cs1 ← setsToMerge.pop()
        cs2 ← setsToMerge.pop()
        csNew ← Merge(cs1, cs2)  ▷ merge sets cs1 & cs2
        setsToMerge.push(csNew)
    end while
    finalSet ← setsToMerge.pop()  ▷ final set of clusters
end procedure
```

3.3 Discussion

With a complete[2] transitivity function, ClusterMerge will not miss any similar element pairs because all elements are implicitly compared against each other, either directly or implicitly via a transitive representative. The chosen element remains representative of its cluster until it is (possibly) fully merged with another cluster. After that, transitivity ensures that subsequent similar elements will then also be similar to the new representative. Therefore, even though cluster members are not necessarily *transitively* represented by the cluster representative, the algorithm also ensures that those non-transitively similar elements retain their own cluster.

In reality, a complete and computationally efficient transitivity function rarely exists for non-trivial elements, so approximation is necessary, as for our motivating example of protein sequence clustering. Incompleteness in the transitivity function can lead ClusterMerge to miss some similar pairs. However, as is demonstrated in §6, even an approximate transitivity function can produce very good results. This is also why transitivity is tested both ways in Algorithm 1, since approximate transitivity is not necessarily symmetric.

The threshold value T is a parameter that would be chosen by an end user or domain expert to specify the desired degree of similarity between elements. Users do not currently have influence over which elements are used as representatives, which are selected by the algorithm.

[2] A *complete* transitivity function correctly captures all transitive similarity in the data.

3.4 Complexity

The worst-case complexity of ClusterMerge is $O(n^2)$, however this is a fairly strict upper bound. Consider the tree structure formed by the cluster set merges, which has a depth of $log_2 n$, where n is the number of elements to be clustered. At the first layer, there are $n/2$ merges possible, each comparing two clusters of one element each. At there second layer, there are $n/4$ merges, each comparing a worst-case total of 4 clusters (if no full clusters were merged in the layer above). Generalizing this pattern we obtain

$$n/2 \times 1^2 + n/4 \times 2^2 + n/8 \times 4^2 \ldots$$

which we can reduce to

$$2n \sum_{i=0}^{log_2 n} 2^i = 2n \cdot 2^{log_2 n+1} - 1 = 2n \cdot (2n) - 1 \approx n^2$$

However, when clusters are fully merged, there is a reduction in work at each level, leading to sub-n^2 performance. In a more optimal case, assuming that at each step the merger of two cluster sets cuts the total number of clusters in half, complexity falls to $O(n\log n)$. Actual complexity therefore depends on the amount of transitivity in the data being clustered.

4 Protein Sequence Clustering

Our motivation for this work is the problem of precise clustering for protein sequences. Given their importance in biology, many specific algorithms for clustering sequences have been developed (§2). Fast algorithms however trade precision for speed and are not able to find a sufficient fraction of the similar sequence pairs in a dataset.

"Similar" sequences in this domain indicate *homologous* sequences/proteins/genes, with homology denoting the existence of a common ancestry between the sequences [16]. Homology within and across genomes can thus be used to propagate protein function annotations [3, 8]. In addition, homologous sequences can aid in the construction of phylogenetic species trees and the study of gene evolution [1]. As explained in §1, databases of homologous proteins are typically constructed using an expensive, all-against-all computation. ClusterMerge can find a set of homologous pairs of proteins that closely approximates the set found by a full all-against-all approach, but at a much lower computational cost.

To apply ClusterMerge, we require a similarity function f, a clustering threshold T, and a transitivity function R. The most accurate similarity function for proteins is a dynamic programming string matching algorithm, typically Smith-Waterman (S-W) [20], which is quadratic in the sequence length. The *score* produced by S-W is a weighted sum of matches, mismatches, and gaps in the optimal alignment, with the weights determined by a substitution matrix. Our clustering threshold will be the same as in Wittwer et al. [27], a S-W score of 181 using the PAM250 substitution matrix [6]. The transitivity function is constructed in a similar way to the incremental greedy clustering [27] and merits some additional explanation.

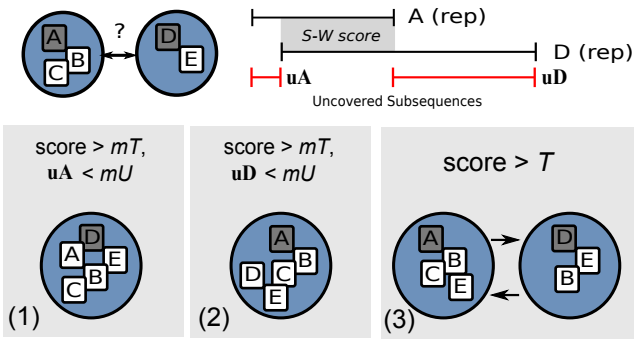


Figure 2. Transitivity function illustration for protein sequence clustering with ClusterMerge.

Protein sequence alignment does have a transitive property, however S-W is a *local* alignment algorithm, meaning that it may not include or "cover" all residues (individual amino acids) in both sequences, especially when the sequences are of different lengths. If a sequence is clustered with a representative that does not completely cover it when aligned, the uncovered subsequence will go unrepresented. This may cause *subsequence homologies* to be missed.

Therefore, subsequence homologies must be taken into account when designing a transitivity function for proteins. Figure 2 illustrates the transitivity function we use, through the example of merging two clusters with representative sequences A and D. Depending on the size of each sequence and the alignment, there may be a number of uncovered residues in each sequence, shown as **uA** and **uD** in Figure 2. To fully merge the clusters, the alignment score between A and D must be greater than mT, the full merge threshold, a parameter. In addition, the number of uncovered residues in one of the sequences must be less than parameter mU (*maximum uncovered*), to ensure that homologous subsequences are not missed. For example, representative D has a large uncovered subsequence in Figure 2, which representative A would not be able to transitively represent. Transitivity does not apply in this case. However, D nearly completely covers A, and assuming **uA** is less than mU, transitivity would apply and D could transitively represent A. The cluster of A would then be fully merged into the cluster of D, with D representing all sequences (situation (1) in Figure 2).

5 Parallel ClusterMerge

There are several opportunities for parallelism inherent in ClusterMerge, which we will use to construct efficient systems for both shared-memory and distributed environments.

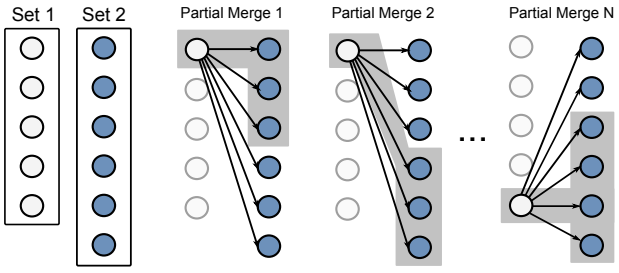


Figure 3. A merge of two large cluster sets is split into partial merges. Threads (or remote workers) can then simultaneously process a merge of two sets.

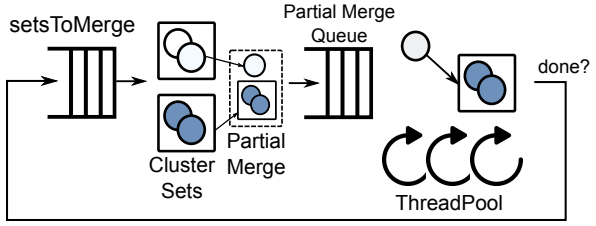


Figure 4. High-level architecture of Shared-CM.

Since the designs for shared-memory and distributed systems differ slightly, we will refer to the shared-memory design as Shared-CM and the distributed design as Dist-CM.

The obvious parallelism in ClusterMerge is that smaller sets near the bottom of the tree can be merged in parallel. In general, as long as there are sets of clusters to be merged in the setsToMerge queue, threads can pop two sets, merge them, and push the result back onto the queue. These operations are independent and can be processed in parallel.

However, after many merges, only a few large sets remain. The "tree-level" parallelism is no longer sufficient to keep system resources occupied, and, in fact, the final set merge is always sequential. Therefore, merges of individual sets must be parallelized, which is also necessary because the sets can grow to be very large.

Shared-CM and Dist-CM both use the same technique to split large set merges into smaller work items called *partial merges*. Consider merging two cluster sets, *Set 1* and *Set 2* (Figure 3). A *partial merge* merges a single cluster from *Set 1* into a subset of the clusters of *Set 2*. Threads or remote workers can execute these partial merges in parallel by running the full inner loop of Algorithm 1. This allows the system to maintain a consistent work granularity by scheduling a similar number of element comparisons in each partial merge. Load is then evenly balanced, preventing stragglers and leading to better efficiency. Shared-CM and Dist-CM differ only in how they coordinate synchronization of the results of partial merges.

5.1 Shared-Memory

Shared-CM is designed to be run on a typical commodity multicore computer. Shared-CM splits set merges into partial merges as described above and allows threads to update the clusters in each set in place.

Consider a thread executing a partial merge, where a cluster from *Set 1* is being merged into some clusters in *Set 2*. While our thread has exclusive access to the cluster from *Set 1*, it has no such guarantee for the clusters in *Set 2*. Concurrent modifications, including removal of clusters and creation of new ones, can happen because of partial merges of other items from *Set 1*.

Shared-CM uses locking to prevent races. The merging logic is the same as in Algorithm 1, however clusters of the second set are locked before being modified in-place. The final merged set is simply the remaining clusters of *Set 1* and *Set 2* that have not been fully merged. Ordering is not guaranteed and the process sacrifices determinism, but the significant pair recall is the same as a deterministic execution.

Figure 4 illustrates the system design. A coordinating thread pops two sets off the *setsToMerge* queue. The merge is divided into partial merges as described above, which are inserted into a partial merge queue. A pool of worker threads then process the partial merges. Once the partial merges for a set merge are completed, the coordinating thread collects remaining clusters from both sets into a merged set and pushes it onto the queue.

As long as there are sets remaining to be merged, partial merges can be scheduled and all processors on the machine kept busy. Multiple cluster set merges can also be split into partial merges and executed simultaneously. Shared-CM can scale nearly linearly across cores, with full experimental results detailed in § 6.2.

5.2 Distributed

While locking works well in a multicore computer, it would limit scalability on a distributed cluster. Instead, Dist-CM ensures that any processing sent to remote workers is fully independent. Workers therefore have no communication with each other and only communicate with a central controller to get more work, resulting in a very scalable system. Dist-CM is a controller-agent distributed system. The *controller* is responsible for managing the shared state of the computation, while the majority of the computing is performed by *remote workers*.

Dist-CM uses several techniques to control the size of an average work item to prevent load imbalance and enable efficient scaling. First, batching is used to group small cluster sets together as a single work item. This provides each remote worker with a unit of computation that will not be dwarfed by communication overhead. Batches are executed by a remote worker, and the resulting cluster set is returned to the controller and pushed back into the *setsToMerge* queue.

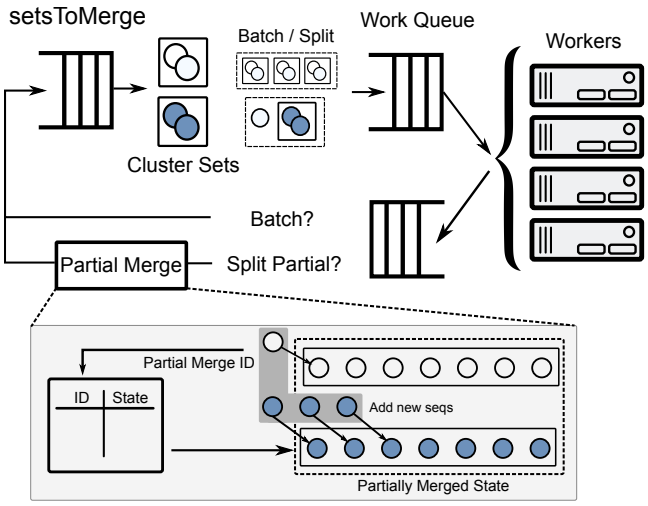


Figure 5. High-level architecture of Dist-CM.

Batching is important for the early phase of computation, where each set is small and requires little computation.

For larger merges near the top of the tree, Dist-CM uses partial merges in much the same manner as Shared-CM to maintain a consistent work item granularity. Because there is no inter-worker communication, the controller is responsible for managing partial merge results as they are returned. Recall that each partial merge work item merges a single cluster from *Set 1* into a subset of clusters of *Set 2*. The result of a partial merge executed by a remote worker is then a set containing some clusters of *Set 2*, with the single cluster from *Set 1* possibly fully merged with one of them and/or some elements exchanged with some clusters. If the single cluster was not fully merged, it will be included in the returned set.

For each outstanding merger of two cluster sets, the controller maintains a partially merged state of the final result, identified by an ID associated with all partial merges involved in its computation. This partially merged state begins as simply both sets of clusters. When a partial merge result is returned to the controller, it uses the ID to look up the associated partially merged state. The controller will then update the partially merged state with the results of the returned partial merge, adding any elements to existing clusters and marking any fully merged clusters. After processing the final partial merge for a given set merge, the merge is complete and the resulting set is constructed by simply combining non-fully merged clusters from both sets.

Figure 5 summarizes the design of Dist-CM. Once again the *setsToMerge* queue is loaded with single element cluster sets, but at the central controller. A coordinating thread on the controller will pop two sets off the queue to merge together. If the sets (in terms of total clusters) are smaller than a batch size parameter, the thread will pop more sets until it has sets whose total number of clusters is equal to or greater than the batch size parameter. These sets are compiled into a batch work item and pushed into a central work queue. If the sets popped by the coordinating thread are large, they are split into partial merges, with the number of sequences in each cluster taken into account to evenly size each request. This dynamic load balancing keeps straggling in remote workers to a minimum, and is important in achieving good scaling. Partial merges are then are pushed into the central work queue as individual work items. The central work queue feeds a set of remote worker nodes.

Results from workers are returned to the controller and either pushed back to the *setsToMerge* queue if a batch result (which is a complete cluster set), or used to update an associated partially merged state if it was a partial merge result. If the partially merged state was completed by the work item in question, the now-complete set is pushed to the *setsToMerge* queue. The process is complete when the final set of the merge tree is complete.

The trade-off inherent in this design is that Dist-CM does more work than necessary in exchange for zero communication among workers. A cluster in a partial merge will continue to be merged into clusters in the set by Dist-CM even if they were fully merged away in other workers. As a result, Dist-CM can perform slightly more work than Shared-CM and can occasionally add the same elements to the same cluster (these duplicates are easily removed by the controller). This trade-off leads to Dist-CM being about 17% slower than Shared-CM when using a single remote worker in our application of protein sequence clustering.

Scalability can be adversely affected by average latency or amount of work in a single work item. Very small work items will have communication overheads that may dwarf the actual computation. Very large work items can cause stragglers and load imbalance that can leave processors idle. Early versions of Dist-CM operated without dynamic sizing of partial merges; each one merged a single cluster into the entirety of the other set. This led to massive load imbalance and long idle periods as large merges were completed. Dynamic sizing of partial merges was crucial to ensure proper load balance and minimize stragglers, improving scaling efficiency by almost 3×.

Furthermore, unbalanced work distribution can cause stragglers as well, if some workers locally queue more work than others. To avoid this, we switched from a round-robin work distribution method to a Join-Idle-Queue [13] approach in which workers inform the controller when they need more work. This keeps all workers busy so long as work is available, while limiting worker-local queuing.

5.3 Optimizations

Several important optimizations enable efficient scaling of Dist-CM. In early versions, the controller sent whole sets with each partial merge, which nearly saturated available network bandwidth. Communication overhead was greatly

reduced through several techniques. First, each worker replicates the sequence dataset and refers to sequences by a 4 byte index. Actual sequence data is never transferred, and even large clusters with thousands of sequences only require a few kilobytes. Second, one of the sets in a series of partial merges is cached on each worker, so it is only transferred over the network once. Finally, the results of partial merge are returned as diffs, i.e., only newly added sequences in each cluster.

6 Evaluation

We evaluate several aspects of ClusterMerge and its implementations applied to protein sequence clustering. Both Shared-CM and Dist-CM variants are evaluated in this section. Both implementations are written in C++ and compiled with GCC 5.4.0. To compute sequence similarities, we use the Smith-Waterman library SWPS3 [23].

We use two datasets for our evaluation. One is a dataset of 13 bacterial genomes extracted from the OMA database [2], a total of 59013 protein sequences (59K dataset). This is the same dataset used by Wittwer et al., which allows comparison with their implementation. The second dataset is a large set of eight genomes from the QfO benchmark totaling 90557 sequences (90K dataset). Although these are a small fraction of the available databases, each represents billions of possible similar pairs, taking many hours to evaluate in a brute-force manner.

Our tests are performed using servers containing two Intel Xeon E5-2680v3 running at 2.5 GHz (12 physical cores in two sockets, 48 hyperthreads total), 256 GB of RAM, running Ubuntu Linux 16.04. The distributed compute cluster consists of 32 servers (768 cores), a subset of a larger, shared deployment. These are connected via 10 Gb uplinks to a 40GbE-based IP fabric with 8 top-of-rack switches and 3 spine switches. The dataset is small enough such that a local copy can be stored on each server. In fact, even large protein datasets are easily stored on modern servers. For example, the complete OMA database of 14 million protein sequences fits within 10GB, a fraction of modern server memory capacity.

Our baseline for clustering comparisons is the incremental greedy precise clustering of [27], which is the only clustering method that can achieve an equivalent level of similar pair recall.

6.1 Clustering and Similar Pair Recall

For consistency, our clustering threshold is the same as the incremental greedy precise clustering in Wittwer et al. [27], a Smith-Waterman score of 181. The threshold is low, but this is necessary to find distant homologs. After ClusterMerge identifies clusters, an intra-cluster, all-against-all comparison is performed, in which the sequence pairs within a cluster are aligned using Smith-Waterman. Those with a score higher than the clustering threshold are recorded as a similar pair. For our datasets, the number of actual similar pairs is small compared to the number of potential similar pairs (e.g. 1.2 million actual versus 1.74 billion potential), leading to relatively few alignments to complete this stage. Biologists may perform additional alignments to derive an optimal alignment with respect to different scoring matrices, however this is orthogonal to the concerns of this paper.

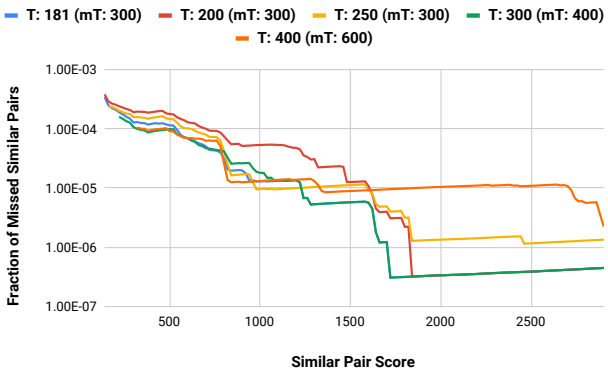


Figure 6. Cumulative fraction of missed pairs reaching at least a certain similarity score, as the clustering threshold T and fully merge threshold mT are varied ($mU = 15$). ClusterMerge shows a low sensitivity to small parameter variations, while most missed similar pairs remain low-scoring ones.

Recall is the percentage of ground truth pairs found by our systems. Ideal recall is 100%. Both Shared-CM and Dist-CM ClusterMerge, using a minimum full merge score (mT) of 250 and a max uncovered residues (mU) of 15, produce clusters with a recall of 99.8 ± 0.01%. Recall variability is negligible and is due to the non-determinism of parallel execution. Of the pairs missed by ClusterMerge, very few were high scoring pairs. The median score of a missed pair is 191 and the average score of a missed pair is 235. These values are very close to the cluster threshold itself (in contrast to high scoring pairs, which can be greater than 1000), indicating that these are not likely biologically "important" pairs (Figure 6). ClusterMerge misses only a handful of high scoring pairs, around one millionth of total significant pairs, as seen in Figure 6.

In clustering the 59K sequence dataset, ClusterMerge performs approximately 871 million comparisons. By contrast, the full, all-against-all comparison requires approximately 1.74 billion comparisons, showing that ClusterMerge reduces comparisons by nearly 50%.

In terms of the clusters themselves, ClusterMerge generates a similar clustering profile as incremental greedy clustering [27], with a total of 33,562 clusters. In each, the vast majority of clusters contain between 1 and 4 sequences, with a few large clusters (33% of clusters contain more than 10 sequences, 8% of clusters contain more than 100 sequences,

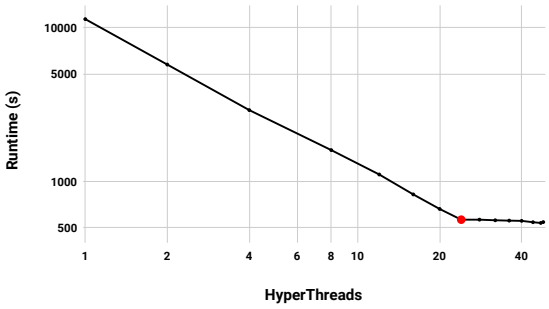


Figure 7. Scaling of Shared-CM up to 48 threads. Scaling is nearly linear up to all 24 physical cores, while hyperthreading provides no benefit.

0.5% of clusters contain more than 1000 sequences). ClusterMerge generates slightly larger outliers, with its largest cluster containing approximately 1500 sequences as opposed to the greedy method's largest cluster of around 1150 sequences.

Figure 6 shows that ClusterMerge and our transitivity function are relatively insensitive to parameter variations. Lower clustering thresholds T and lower full merge thresholds mT generally lower the number of missed similar pairs, although the absolute percentage of missed pairs remains extremely low, with the majority being low-scoring pairs.

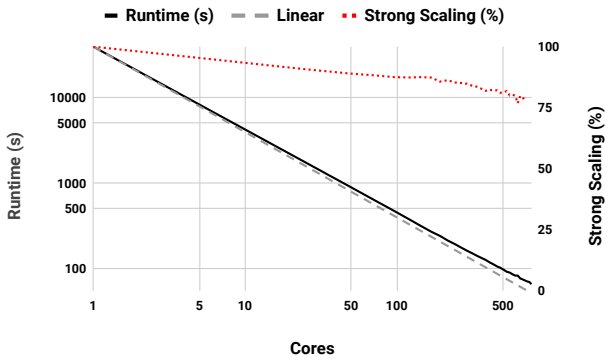
6.2 Multicore Shared-Memory Performance

In this section, we evaluate how well Shared-CM performs on a single multicore node. This experiment uses a reduced dataset of 28600 sequences, to reduce runtimes at low thread counts. Figure 7 shows the total runtime decreases as we increase the number of threads. Shared-CM achieves near linear scaling — profiling with Intel VTune indicates little or no lock contention. Memory access latency and NUMA have no effect as the workload is compute bound.

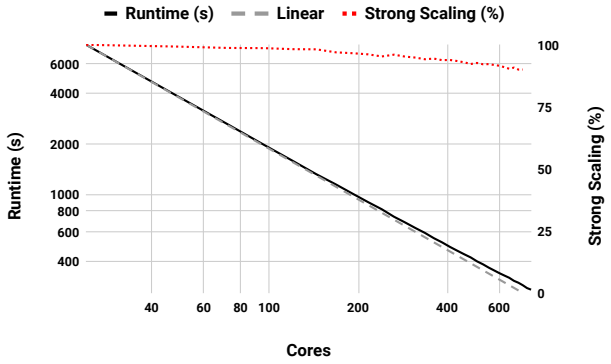
Note, however, that scaling is linear only on physical cores. The primary compute bottleneck is the process of aligning representative sequences using Smith-Waterman, which processes data that fits in the L1 cache and is able to saturate functional units with a single thread. Therefore, hyperthreading provides no benefit.

The only major impediment to perfect scaling is some loss of parallelism before the last and second-last merges, since the second-last merge must be fully completed before work for the last merge can start to be scheduled.

Shared-CM with a single thread clusters the bacteria dataset in 31905 seconds, compared to 1486 seconds with 24 threads, a speedup of 21.5×. To compare with incremental greedy clustering, we run Wittwer's single-threaded code [27] on our machine with the same dataset, resulting in a runtime of 89486 seconds. Shared-CM is approximately 2.8× faster on a single core, and 60.2× faster using all cores.



(a) 59K dataset, 79% scaling efficiency.



(b) 90K dataset, 90% scaling efficiency.

Figure 8. Scaling of Dist-CM over 32 servers (768 cores).

6.3 Distributed Performance

Dist-CM allows us to scale ClusterMerge beyond a single server. To evaluate the scaling of Dist-CM, we hold the dataset size constant and vary the number of servers used to process work items (batches or partial merges), otherwise known as *strong scaling*. The baseline single core runtime for Dist-CM clustering the 59K dataset is 39314 seconds. Figure 8a shows that on 32 nodes (768 cores) Dist-CM clusters the dataset in 65 seconds, resulting in a speedup of 604×. Strong scaling efficiency at 768 cores is 79%. Compared to single-threaded incremental greedy clustering [27], Dist-CM is 2.27× faster using a single core, and 1400× faster using the full compute cluster.

The reason for sublinear scaling is essentially the same as with Shared-CM — around the last few merges of cluster sets, work scheduling may halt as the system waits for an earlier merge to finish before being able to schedule more work. There will always be some small portion of sequential execution, so perfect scaling is impossible by Amdahl's Law.

That being said, this sequential section is proportionally lower with larger datasets. Figure 8b shows Dist-CM strong scaling when clustering the larger 90K sequence dataset.

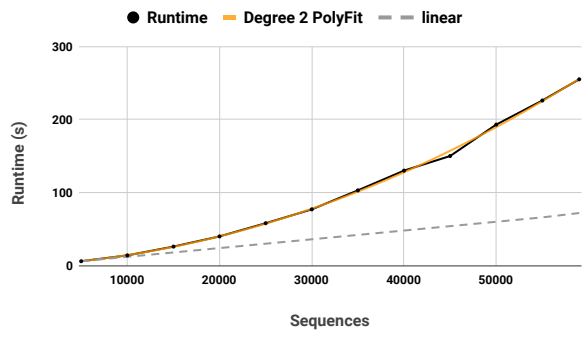


Figure 9. Workload scaling of Dist-CM.

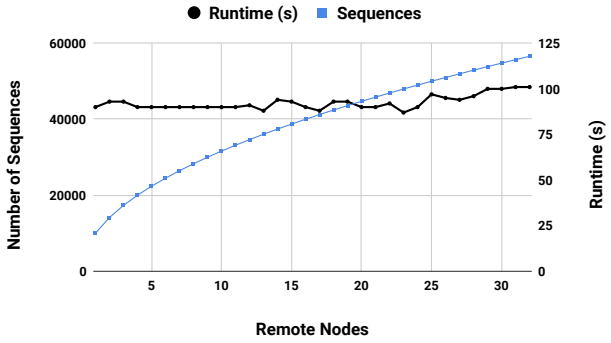


Figure 10. Dist-CM weak scaling over a 32 node (768 core) cluster. Nearly 100% efficiency at 32 nodes.

The scaling is much more efficient (90% at 32 nodes), with a speedup of 28.7× relative to one server node.

In addition, we perform a weak scaling experiment in which we vary the amount of work in proportion to the number of nodes. Because our dataset is evolutionarily diverse and has relatively low levels of transitivity, ClusterMerge is closer to $O(n^2)$ in the number of sequences. The amount of actual work increases quadratically with the number of sequences. Figure 9 clearly shows this by varying the number of sequences that Dist-CM clusters using 10 worker nodes. The runtime curve fits almost exactly to a degree two polynomial. Therefore, for our weak scaling experiment, we vary the number of sequences at each step by a square root factor to maintain a proportional increase in workload. Figure 10 shows the results, again while clustering using 1 to 32 nodes. Runtime remains nearly constant throughout, indicating a weak scaling efficiency of 95-100%. We thus expect that Dist-CM will be able to cluster much larger datasets while maintaining high scaling efficiency.

6.4 Effect of Dataset Composition

As noted in section §3.4, complexity and therefore runtime depend on how many clusters can be fully merged at each level of the tree. If the transitivity function accurately represents similar elements, the number of full merges at each level is primarily affected by the number of transitively similar elements in a dataset. More transitively similar elements will result in more complete cluster merges, bringing runtime complexity closer to the $O(n\log n)$ optimum.

For protein clustering, the dataset with 13 bacterial genomes has a relatively low number of transitively similar sequences since the species are genetically very distant (more distant than human and plants). Given a set of more closely related genomes, with more transitively similar sequences, we would expect ClusterMerge to generate fewer clusters and run much faster. To test this hypothesis, we clustered a third dataset of more closely related *Streptococcus* bacteria genomes, consisting of 33 genomes (69648 sequences, similar to the other dataset).

Using Shared-CM with 48 threads, the clustering is completed in 283 seconds, producing 10500 clusters. As predicted, clustering is much faster than the 13 bacterial genome dataset (1486 seconds) as the number of clusters is much lower. In addition, ClusterMerge again produced a high recall of 99.7% of similar pairs relative to a full all-against-all.

7 Future Work

Both of our implementations, Shared-CM and Dist-CM, perform and scale well. However many improvements are possible. Dataset size may expose limits to the current implementation. Very large clusters may produce work items that are still too large, which may cause straggling. Additional splitting beyond the current partial merge may be necessary. Extreme imbalance in cluster sizes between two sets to be merged may also require more creative scheduling of partial merges to avoid large variation in work item size.

For our application to proteins, the current computational bottleneck is the Smith-Waterman alignment function. Runtimes could be improved with a more efficient S-W implementations. We are actively investigating protein alignment-friendly S-W hardware implementations. Similarly, more approximate or less precise alignment methods could be used, though this may come at the cost of precision.

8 Conclusion

ClusterMerge is a parallel and scalable algorithm for precise clustering of elements. When applied to protein sequences, ClusterMerge produces clusters that encompass 99.8% of significant pairs found by a full all-against-all comparison, while performing 50% fewer similarity comparisons. Our implementations achieve speedups of 21.5× on a 24-core shared-memory machine and 604× on a cluster of 32 nodes (768 cores). The distributed implementation of ClusterMerge for protein clustering can produce clusters 1400× faster than a single-threaded greedy incremental approach. ClusterMerge is open source and available [25].

Our hope is that ClusterMerge will help to form a comprehensive "map" of protein sequences. In theory, clustering could proceed to a point where any given new protein sequence would be represented completely by a subset of existing clusters. No new clusters would need to be added, and any new protein could be classified in $O(n\log n)$ time only. However, it is not yet clear how many different genomes would be required to form such a map.

References

[1] Adrian M. Altenhoff and Christophe Dessimoz. 2012. Inferring Orthology and Paralogy. In *Evolutionary Genomics: Statistical and Computational Methods, Volume 1*, Maria Anisimova (Ed.). Humana Press, Totowa, NJ, 259–279. https://doi.org/10.1007/978-1-61779-582-4_9

[2] Adrian M. Altenhoff, Natasha M. Glover, Clément-Marie Train, Klara Kaleb, Alex Warwick Vesztrocy, David Dylus, Tarcisio M. de Farias, Karina Zile, Charles Stevenson, Jiao Long, Henning Redestig, Gaston H. Gonnet, and Christophe Dessimoz. 2018. The OMA orthology database in 2018: retrieving evolutionary relationships among all domains of life through richer web and programmatic interfaces. *Nucleic Acids Research* 46, D1 (Jan. 2018), D477–D485. https://doi.org/10.1093/nar/gkx1019

[3] Adrian M. Altenhoff, Romain A. Studer, Marc Robinson-Rechavi, and Christophe Dessimoz. 2012. Resolving the Ortholog Conjecture: Orthologs Tend to Be Weakly, but Significantly, More Similar in Function than Paralogs. *PLOS Computational Biology* 8, 5 (May 2012), e1002514. https://doi.org/10.1371/journal.pcbi.1002514

[4] Bill Andreopoulos, Aijun An, Xiaogang Wang, and Michael Schroeder. 2009. A roadmap of clustering algorithms: finding a match for a biomedical application. *Briefings in Bioinformatics* 10, 3 (May 2009), 297–314. https://doi.org/10.1093/bib/bbn058

[5] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/304182.304187

[6] M. O. Dayhoff and R. M. Schwartz. 1978. Chapter 22: A model of evolutionary change in proteins. In *in Atlas of Protein Sequence and Structure*.

[7] Robert C. Edgar. 2010. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics* 26, 19 (Oct. 2010), 2460–2461. https://doi.org/10.1093/bioinformatics/btq461

[8] Pascale Gaudet, Michael S. Livstone, Suzanna E. Lewis, and Paul D. Thomas. 2011. Phylogenetic-based propagation of functional annotations within the Gene Ontology consortium. *Briefings in Bioinformatics* 12, 5 (Sept. 2011), 449–462. https://doi.org/10.1093/bib/bbr042

[9] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. 2000. Rock: A robust clustering algorithm for categorical attributes. *Information Systems* 25, 5 (July 2000), 345–366. https://doi.org/10.1016/S0306-4379(00)00022-3

[10] Maria Hauser, Christian E. Mayer, and Johannes Söding. 2013. kClust: fast and sensitive clustering of large protein sequence databases. *BMC Bioinformatics* 14, 1 (Aug. 2013), 248. https://doi.org/10.1186/1471-2105-14-248

[11] S Henikoff and J G Henikoff. 1992. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America* 89, 22 (Nov. 1992), 10915–10919. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC50453/

[12] Weizhong Li and Adam Godzik. 2006. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics (Oxford, England)* 22, 13 (July 2006), 1658–1659. https://doi.org/10.1093/bioinformatics/btl158

[13] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. 2011. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation* 68, 11 (Nov. 2011), 1056–1071. https://doi.org/10.1016/j.peva.2011.07.015

[14] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. 2016. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology* 17, 1 (June 2016), 132. https://doi.org/10.1186/s13059-016-0997-x

[15] Rafail Ostrovsky and Yuval Rabani. 2007. Low Distortion Embeddings for Edit Distance. *J. ACM* 54, 5 (Oct. 2007). https://doi.org/10.1145/1284320.1284322

[16] C. Patterson. 1988. Homology in classical and molecular biology. *Molecular Biology and Evolution* 5, 6 (July 1988), 603–625. https://doi.org/10.1093/oxfordjournals.molbev.a040523

[17] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-set Data Structure. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 62:1–62:11. http://dl.acm.org/citation.cfm?id=2388996.2389081

[18] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2013. Scalable Parallel OPTICS Data Clustering Using Graph Algorithmic Techniques. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, 49:1–49:12. https://doi.org/10.1145/2503210.2503255

[19] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 1998. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Mining and Knowledge Discovery* 2, 2 (June 1998), 169–194. https://doi.org/10.1023/A:1009745219419

[20] T. F. Smith and M. S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (March 1981), 195–197. https://doi.org/10.1016/0022-2836(81)90087-5

[21] Erik L. L. Sonnhammer, Toni Gabaldón, Alan W. Sousa da Silva, Maria Martin, Marc Robinson-Rechavi, Brigitte Boeckmann, Paul D. Thomas, and Christophe Dessimoz. 2014. Big data and other challenges in the quest for orthologs. *Bioinformatics* 30, 21 (Nov. 2014), 2993–2998. https://doi.org/10.1093/bioinformatics/btu492

[22] Martin Steinegger and Johannes Söding. 2018. Clustering huge protein sequence sets in linear time. *Nature Communications* 9, 1 (June 2018), 2542. https://doi.org/10.1038/s41467-018-04964-5

[23] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. 2008. SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and ×86/SSE2. *BMC Research Notes* 1, 1 (Oct. 2008), 107. https://doi.org/10.1186/1756-0500-1-107

[24] Richard Van Noorden, Brendan Maher, and Regina Nuzzo. 2014. The top 100 papers. *Nature News* 514, 7524 (Oct. 2014), 550. https://doi.org/10.1038/514550a

[25] EPFL VLSC. 2019. ClusterMerge Repository. https://github.com/epfl-vlsc/clustermerge

[26] Robert M. Waterhouse, Fredrik Tegenfeldt, Jia Li, Evgeny M. Zdobnov, and Evgenia V. Kriventseva. 2013. OrthoDB: a hierarchical catalog of animal, fungal and bacterial orthologs. *Nucleic Acids Research* 41, D1 (Jan. 2013), D358–D365. https://doi.org/10.1093/nar/gks1116

[27] Lucas D. Wittwer, Ivana Piližota, Adrian M. Altenhoff, and Christophe Dessimoz. 2014. Speeding up all-against-all protein comparisons while maintaining sensitivity by considering subsequence-level homology. *PeerJ* 2 (Oct. 2014), e607. https://doi.org/10.7717/peerj.607