CD-HIT: accelerated for clustering the next generation sequencing data

Limin Fu, Beifang Niu, Zhengwei Zhu†, Sitao Wu and Weizhong Li*

Center for Research in Biological Systems, University of California San Diego, La Jolla, California, USA

ABSTRACT

Summary: CD-HIT is a widely used program for clustering biological sequences to reduce sequence redundancy and improve the performance of other sequence analyses. In response to the rapid increase in the amount of sequencing data produced by the next generation sequencing technologies, we have developed a new CD-HIT program accelerated with a novel parallelization strategy and some other techniques to allow efficient clustering of such datasets. Our tests demonstrated very good speedup derived from the parallelization for up to ~24 cores and a quasi-linear speedup for up to ~8 cores. The enhanced CD-HIT is capable of handling very large datasets in much shorter time than previous versions.

Availability: http://cd-hit.org
Contact: liwz@sdsu.edu
Supplementary information: Supplementary data are available at Bioinformatics online.

1 INTRODUCTION

Sequence analysis has played a crucial role in computational biology. With the advancement of the next generation sequencing technologies, the amount of available sequencing data is growing exponentially. Removing redundancy from such data by clustering could be crucial for reducing storage space, computational time and noise interference in some analysis methods etc.

CD-HIT was originally developed to cluster protein sequences to create reference databases with reduced redundancy (Li, et al., 2001) and was then extended to support clustering nucleotide sequences (Li and Godzik, 2006). Since its release, CD-HIT has become very widely used for a large variety of applications ranging from non-redundant dataset creation (Suzek, et al., 2007), protein family classifications (Yooseph, et al., 2008), artifact identification (Niu, et al., 2010), metagenomics annotation (Sun, et al., 2011), RNA analysis (Loong and Mishra, 2007), to various prediction studies (Rubinstein and Fiser, 2008).

With sequencing data rapidly growing in public data repositories as well as in individual laboratories, there has been strong demand for an enhanced CD-HIT with greater efficiency. In response to such demand, we have developed this enhanced and parallelized version of CD-HIT, to exploit the fact that multi-core machines have become very common in ordinary laboratories.

A computer cluster based parallelization procedure for CD-HIT has been proposed in (Suzek, et al., 2007), though not fully parallelized, this procedure provides good speedup using computer cluster. Since computer clusters are not as easily available as multi-core machines, here we propose an alternative parallelization technique, which assumes shared memory model and works well on multi-core machines.

2 METHODS

Basically CD-HIT is a greedy incremental algorithm that starts with the longest input sequence as the first cluster representative, and then processes the remaining sequences from long to short to classify each sequence as a redundant or representative sequence based on its similarities to the existing representatives. The similarities are estimated by common word counting using word indexing and counting tables to filter out unnecessary sequence alignments, which are used to compute exact similarities. In the following sections, we will describe the techniques that are used to accelerate CD-HIT.

2.1 Simplification

In order to support full parallelization, the core steps of CD-HIT has been simplified into two key procedures: a checking procedure and a clustering procedure. Using these two procedures, the algorithm requires at most two word tables without the need to swap them to disk, which was necessary in the original CD-HIT due to the use of multiple tables for large datasets.

Given a word table, the checking procedure checks each of the remaining sequences against the table, and determine if it is a redundant sequence. The clustering procedure will make a final determination of the status of a sequence, and if it is classified as a representative sequence, it is used to update the word table. A more detailed description with illustration is available in the Supplementary Material Section 1.2 and Figure S1.

2.2 Parallelization

Given T threads or cores, the basic idea of our novel parallelization technique is to use two word tables, and use T-1 threads to run multiple checking procedures using one word table (an immutable checking table), and the remaining thread to run a single clustering procedure using the other table (a mutable clustering table) in parallel. Due to the sequential characteristics of CD-HIT, it will require properly grouping the input sequences and switch the word tables to guarantee the correctness of the parallelization.

This is achieved by dividing each round of computation into two stages. The first stage is to run T checking procedures on the sequence group defined for this round of computation using the word table (checking table) from the previous round. Then the second stage will use an addi-
tional and empty word table (clustering table) to run a clustering procedure in one thread on the current sequence group, and at the same time the remaining 7-1 threads will run multiple checking procedures on the remaining sequences.

Since the clustering procedure may finish before or after the checking procedures, proper scheduling is used to guarantee all threads are active most of the time. At the end of each round, the clustering table will become the checking table of the next round, and the checking table of this round will be emptied to become the clustering table of the next round. More information including detailed description, illustration and pseudo codes etc. are available in the Supplementary Material (Section 1.3 - 1.5).

2.3 Other enhancements

Besides the parallelization, the new CD-HIT includes other enhancements such as faster file reading, better filtering threshold estimation, more efficient word counting and better alignment band estimation etc. The new filtering threshold estimation is slightly more precise and can filter out more unnecessary alignments. Now word counting is handled more efficiently for input datasets with high redundancy, by maintaining a smaller counting array for hit representatives instead of a full counting array for every representatives. The improved alignment band estimation can find a narrower band for banded alignment.

2.4 Implementation

This enhanced CD-HIT is implemented in the C++ programming language, and uses OpenMP (http://www.openmp.org) for parallelization. The parallel for construct of OpenMP is used for running the checking and clustering procedures in multiple threads with dynamic scheduling. Different computation data buffers are allocated for different threads. The checking word table is immutable and shared by multiple threads.

| Dataset   | CD-HIT3 | CD-HIT4 | CD-HIT4 (8 cores) | UCLUST5 |
|-----------|---------|---------|-------------------|---------|
| Swissprot | 80 min  | 58 min  | 12 min            | 15 min  |
| NR        | 44 min  | 22 min  | 6 min             | 46 min  |
| Twinstudy | 47 min  | 19 min  | 4 min             | 56 min  |
| HumanGut  | 494 min | 42 min  | 8 min             | 214 min |

Table 1. Comparison to the previous CD-HIT and UCLUST

UCLUST5 free version cannot run on the full NR, TwinStudy and HumanGut datasets, so subsets with approximately 1M sequences of NR, 1M reads of TwinStudy and 4M reads of HumanGut are used in this comparison.

3 RESULTS

To see how much performance improvement has been achieved, we tested the new CD-HIT (V4.6) and the old CD-HIT (V3.1.2) on a set of datasets including two protein sequence datasets: SWISS-PROT (~0.4M sequences), NR (~12M sequences); and two DNA sequence datasets: HumanGut (MH0002, ~23M reads, Qin, et al., 2010) and TwinStudy (SRP000319, ~8M reads, Turnbaugh, et al., 2009). Both the SWISS-PROT and the NR datasets were downloaded from NCBI (ftp://ftp.ncbi.nih.gov/blast/db/FASTA/) on October 20, 2010. We also compared CD-HIT to a similar program UCLUST (V5.1.221) from (Edgar, 2010). These tests were done on a Debian Linux server with 4 twelve-core AMD Opteron 6172 processors. Equivalent parameters were used for different programs whenever possible. Details and additional tests are available in the Supplementary Material 2.

Table 1 compares the efficiency of the enhanced CD-HIT to the previous version of CD-HIT and the latest UCLUST. The results demonstrate that the new CD-HIT without using multi-core is significantly more efficient than the old one, and is comparable to or more efficient than UCLUST as well. When multi-cores are used, the new CD-HIT is much more efficient than either of them. To test the effectiveness of the parallelization in the new CD-HIT, the full datasets were clustered using different number of cores. The time speedups are shown in Figure 1, which indicates the parallelization has good speedup up to ~24 cores with a quasi-linear speedup for up to ~8 cores. Besides speed improvements, the new CD-HIT also has better clustering quality than the old CD-HIT and UCLUST (Supplementary Material and Table S2).

![Fig. 1. Evaluation of CD-HIT parallelization: computational time speedup with respect to the number of used CPU cores](http://bioinformatics.oxfordjournals.org/)

4 CONCLUSIONS

In this application note, we presented an enhanced CD-HIT that has been accelerated by a novel parallelization technique and a few other improvements. We tested and demonstrated this new CD-HIT can achieve significant speedup over the previous CD-HIT using a single core, and its acceleration by multi-core computer can scale up well to a reasonable number of cores. Clustering on large datasets that normally runs for days can now finish in hours on multicore machines. We believe this enhanced CD-HIT will find more applications in handling the next generation sequencing data.

ACKNOWLEDGEMENTS

Funding: This study was supported by Award R01RR025030 from the National Center for Research Resources and R01HG005978 from the National Human Genome Research Institute.

REFERENCES

Edgar, R.C. (2010) Search and clustering orders of magnitude faster than BLAST, Bioinformatics.

Li, W. and Godzik, A. (2006) Cdd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences, Bioinformatics, 22, 1658-1659.
Li, W., Jaroszewski, L. and Godzik, A. (2001) Clustering of highly homologous sequences to reduce the size of large protein databases, *Bioinformatics*, **17**, 282-283.

Loong, S.N.K. and Mishra, S.K. (2007) Unique folding of precursor microRNAs: Quantitative evidence and implications for de novo identification, *RNA-a Publication of the RNA Society*, **13**, 170-187.

Niu, B., et al. (2010) Artificial and natural duplicates in pyrosequencing reads of metagenomic data, *BMC Bioinformatics*, **11**, 187.

Qin, J., et al. (2010) A human gut microbial gene catalogue established by metagenomic sequencing, *Nature*, **464**, 59-65.

Rubinstein, R. and Fiser, A. (2008) Predicting disulfide bond connectivity in proteins by correlated mutations analysis, *Bioinformatics*, **24**, 498-504.

Sun, S., et al. (2011) Community cyberinfrastructure for Advanced Microbial Ecology Research and Analysis: the CAMERA resource, *Nucleic Acids Res.*, **39**, D546-551.

Suzek, B.E., et al. (2007) UniRef: comprehensive and non-redundant UniProt reference clusters, *Bioinformatics*, **23**, 1282-1288.

Turnbaugh, P.J., et al. (2009) A core gut microbiome in obese and lean twins, *Nature*, **457**, 480-U487.

Yooseph, S., Li, W. and Sutton, G. (2008) Gene identification and protein classification in microbial metagenomic sequence data via incremental clustering, *BMC Bioinformatics*, **9**, 182.
Supplementary Information: CD-HIT: accelerated for clustering the next generation sequencing data
Limin Fu, Beifang Niu, Zhengwei Zhu, Sitao Wu, Weizhong Li*
Center For Research in Biological Systems, University of California, La Jolla, CA, USA
* E-mail: liwz@sdsc.edu

1 Algorithm Description and Illustration

1.1 Basic CD-HIT Algorithm

The basic CD-HIT algorithm sorts the input sequences from long to short, and processes them sequentially from the longest to the shortest. The first sequence is automatically classified as the first cluster representative sequence. Then each query sequence of the remaining sequences is compared to the representative sequences found before it, and is classified as redundant or representative based on whether it is similar to one of the existing representative sequences.

Such comparison is done in an efficient way by using word counting and indexing table to filter out most of the unnecessary comparisons by a filtering process. For each word from the query sequence, this process retrieves a list of representative sequences that contain the word hits using the word table, and in the mean time, the count of common words between the query and each of the retrieved representatives is updated. Then only those representatives with common word counts above certain threshold are passed to an alignment procedure to compute the similarity (alignment identity) between the query and the representative. The alignment procedure also uses word counting techniques to define a proper band to do efficient banded alignment. If the alignment identity satisfies the specified requirement, the query sequence is classified as a representative sequence, and the word table is subsequently updated to include the counts of the words found in this new representative sequence.

1.2 Basic CD-HIT Algorithm with Limited Table Size

When a huge number of sequences are classified as representatives, the word table will become very large and require a massive amount of memory. In the original implementation, the input sequences are divided into groups (segments), which are processed one after another. One word table is generated from the representative sequences of each processed group. The sequence data of each processed group and the associated word table are swapped into the memory from the disk when needed, and swapped out otherwise, to reduce memory usage.

In the new CD-HIT, we redesigned the memory reduction strategy to remove the need for swapping word tables between memory and disk. The idea is to maintain only one active sequence group and construct one word table from the representative sequences obtained by clustering the sequences in the active group. Then the constructed word table is used to process all the remaining sequences so that the sequence group and word table can be discarded or cleared for handling the next sequence group. This strategy of sequence grouping and size-constrained word table construction forms the basics of our parallelized CD-HIT algorithm that will be described in the next sub-section.
To better illustrate the clustering process with grouped sequences, we define two types of procedures: *checking* and *clustering*. Checking is the procedure to check a sequence against a word table to determine if the sequence is redundant. Clustering is the procedure to determine if a sequence is a representative sequence, and if yes, update the word table with the words from this new representative sequence. For the correctness of the algorithm, each sequence has to be checked against the word tables for the previous groups before it can be clustered against the word table of the current group.

So the new basic clustering algorithm looks like the following: the first sequence is clustered as a representative by default and is used to build the word table. The following sequences are

**Figure S1:** Basic CD-HIT algorithm with size-constrained word table. Representative and redundant sequences are marked as red and green respectively. The blocks with light red and green stripes (A-F) represent processed sequences that have been clustered and are no longer needed to process other sequences. The gray blocks are the current group of sequences to be processed by the clustering procedure. The dark red and green strips are the sequences being used to check the rest of the sequences. (A) each new round of clustering procedure starts with the first gray sequence and an empty word table; (B) the clustering procedure will mark new redundant sequences and update the word table with new representative sequences; (C) the clustering procedure is carried out until all the gray sequences in the current group are processed; (D) then the checking procedure starts with the first unmarked sequence; (E) the checking procedure will mark new redundant sequences; (F) the clustering procedure is carried out until all the remaining unmarked sequences are processed; then a new sequence group is defined for the next round of clustering and checking procedures starting from A.
clustered against this table one by one, until the size of the table reaches the predefined limit. Then each of the remaining sequences in the dataset is checked against the table and marked as redundant if necessary. Those marked as redundant will be skipped in the following procedures. After all the sequences have been checked, we clear the word table and repeat the same steps of clustering and checking procedures, starting from the first sequence that has not gone through the clustering procedure. The algorithm stops when all the sequences have been either marked as redundant or processed by the clustering procedure. Figure S1 illustrates the key steps of each clustering and checking cycle in the basic CD-HIT algorithm.

1.3 Parallelization Strategy

The basic algorithm needs only one word table throughout the whole clustering process. But to realize an effective parallelization of the clustering process, we need two word tables, one will be called checking table, the other clustering table. The usage of these two different tables will become clear in the following description of the parallelization technique.

To reduce unparallelized execution, the first group is usually small, and can be processed by the original sequential CD-HIT algorithm to cluster those sequences and build the first word table. Of course, the first group can also be subdivided into even smaller groups, and parallelized within them.

Once we have the first word table, we will take this table as the checking table and create an empty clustering table. Now the following computation can be done in parallel as a two-stage process which is repeated until finish. The first stage will carry out multiple checking procedures in parallel. For this, we define a new group of sequences (lighter gray section in Figure S2 A,B,C), and use multiple threads (green nodes and arrows on the right side of the sub-figures) to check the sequences of this group against the current checking table. Once this stage is done, the second stage will carry out one clustering procedure/thread and one or multiple checking procedure(s)/threads(s) in parallel as the following: the clustering procedure (red node and arrow on the right side of Figure S2 D,E,F) will be carried out on the new group of sequences that have just finished the checking procedures in the first stage; and the checking procedure(s) will check the remaining sequences against the current checking table. The clustering procedure will use and update the clustering table.

The second stage will proceed until all the remaining sequences have been checked against the current checking table. If the clustering thread has finished processing the sequences in the new group, it will switch to do the checking procedure. And if the checking threads finished before the clustering thread, the clustering thread will be terminated to finish the current cycle of parallelization. This will guarantee that all the threads will be doing active computation. Once the second stage is done, the clustering table will become the current checking table, and the previous checking table will be cleared and used as the current clustering table. Then these two stages of computation will be repeated starting from the first sequence that hasn’t been processed by the clustering procedure.
1.4 Pseudo Codes

To better illustrate the parallelization algorithm, we supplement the text description and figures with the following pseudo codes. In the codes, a *PARALLEL FOR* statement means multiple threads can execute the loop body in parallel with the index variable taking a different value. Here we assume each thread has a private copy of the index variable, and is dynamically sched-

![Figure S2: Parallelized CD-HIT algorithm.](image)

The color denotations in this figure are identical to those in Fig. S1. Here the light green and red circles stand for checking threads and clustering threads respectively. The sequence of steps A-F have been rotated with respect to that in Fig. S1 to simplify the illustration of the parallel algorithm. Before starting the first round of step A, it is assumed that a small group of sequences have been already processed by the sequential CD-HIT algorithm to build the first word table that will be used as the checking table in A. (A) redefine the current group of sequences, use all threads to run checking procedures on the current group in parallel; (B) each thread takes one unmarked sequence and check it against the checking table to determine if it is a redundant sequence; (C) the checking procedures are carried out until all the sequences in the current group have been processed; (D) the first (clustering) thread runs the clustering procedure starting from the beginning of the current sequence group; in the mean time, the other (checking) threads run the checking procedures on the remaining sequences in the dataset; (E) the clustering thread detects new representative sequences and uses them to update the clustering table; the checking threads detect and mark new redundant sequences; (F) the clustering and checking procedures are continued until the checking procedures reaches the end of the dataset. The algorithm repeats A through F until all sequences have undergone the clustering procedure.
uled to take the first index value that hasn’t been processed by any thread. Different computation
data buffers should be allocated for different threads, and the checking word table is immutable
and can be shared by multiple threads. The parallelization model proposed here is very suitable
to be implemented with OpenMP for multi-core machines.

-- define a group starting from sequence index "first":
PROCEDURE DefineGroup( first );

-- return the index of the last sequence in the group:
PROCEDURE DefineGroupEnd( first );

-- use "table" to check "sequence":
PROCEDURE Checking( table, sequence );

-- mark "sequence" if it is redundant:
PROCEDURE CheckingEnd( table, sequence );

-- use "table" to cluster "sequence":
PROCEDURE Clustering( table, sequence );

PROCEDURE ClusteringEnd( table, sequence );

PROCEDURE ParallelCDHIT( sequences, N )

-- create table for checking procedures
tabcheck = NEW table;

-- create table for the clustering procedure
tabclust = NEW table;

-- all loops can skip sequences marked as redundant
WHILE first <= N DO

IF tabcheck NOT empty THEN

-- first stage:
PARALLEL FOR j = first...last DO
-- all threads do the checking procedure
Checking( tabcheck, sequences[j] );
END

checked = FALSE;

-- second stage:
PARALLEL FOR j = last...N DO
IF j == last THEN

-- the first thread does clustering
FOR k = first...last DO
Clustering( tabclust, sequences[k] );
IF checked == TRUE THEN
first = k + 1;
BREAK;
END
END
ELSE

-- all other threads do checking
Checking( tabcheck, sequences[j] );
IF j == N THEN checked = TRUE; END
END
END
IF first == 1 OR last == N THEN

-- handle the first and last group
FOR k = first...last DO
Clustering( tabclust, sequences[k] );
END
END

-- clustering table becomes checking table
1.5 Analysis of The CD-HIT Parallelization Algorithm

This parallelization algorithm is very effective for large datasets. In fact, for large datasets, the
time spent in handling the first group and the last group can be neglected, hence the effectiveness
or the speed-up of the parallelization is mostly determined by the two parallel-for loops in the
pseudo codes. But it is clear that the first parallel loop can be perfectly parallelized because
each of the sequences in the group can be checked independently against the checking table. So
the speed-up will be mostly affected by the second parallel loop.

The situation in the second parallel loop is a bit more complicated, because not all the threads
do the same type of computation. Due to the dynamic scheduling of the threads, the first thread
will start with the first index value and execute the inner loop to do the clustering procedure,
while the other threads will loop on the other index values to do the checking procedures. Now it
may happen that it takes longer time for the inner loop to finish than the outer loop to reach its
last index. When this happens, the inner loop will be interrupted to ensure the thread running
the inner loop will not become the only active thread. When the inner loop is interrupted, the
algorithm will start the next round of the two stages of parallel computation starting with the
index where it left the inner loop. But if the inner loop finishes before the outer loop reaches
the last index, the thread running the inner loop will switch to do checking procedures as well.
In this case, all the threads are still all active.

So in the ideal case, all the threads of the two stages of parallel computation will be active.
If the dataset is big enough, the time spent in handling the first and the last group will be
negligible, so the speedup can theoretically approach the number of used CPU cores. However,
the sizes of the checking table and clustering table are determined by the size of the input
dataset, the memory limit specified for the program and the number of used CPU cores. When
sequences are short and the table sizes become small, the overhead of thread context switching
due to the multi-cycle of two-stage parallel computation will become significant enough to reduce
the speedup of the parallelization.

2 Detailed Testing Results

The datasets used for testing are described in the main manuscript.

2.1 Comparisons to The Previous CD-HIT and UCLUST

The results (see Table S1) indicate the new CD-HIT, even without parallelization, is signifi-
cantly faster than the old one in most cases, and produces fewer clusters with the same identity
thresholds, which means the filtering by word table and the sequence similarity assessment in
the new CD-HIT is better than that in the old one. These results did not confirm the original
claim that UCLUST is superior than CD-HIT in terms of speed (Edgar, 2010). In fact, only
in the test with low identity, UCLUST is faster than CD-HIT, while in the other cases it is significantly slower than CD-HIT.

Table S1: Comparison to the previous CD-HIT and UCLUST

| Dataset       | #Sequence | Program              | Time   | #Cluster | Word | Identity¹ |
|---------------|-----------|----------------------|--------|----------|------|-----------|
| SWISS-PROT    | 437,168   | cd-hit (v4.6) 8 cores| 12 min | 147,495  | 4    | 60%       |
|               |           | cd-hit (v4.6)²       | 58 min | 147,495  |      |           |
|               |           | cd-hit (v3.1.2)³     | 80 min | 150,526  |      |           |
|               |           | uclust (v5.1.221)⁴  | 15 min | 147,390  |      |           |
| NR 1M         | 999,216   | cd-hit (v4.6) 8 cores| 6 min  | 655,664  | 5    | 70%       |
|               |           | cd-hit (v4.6)        | 22 min | 655,664  |      |           |
|               |           | cd-hit (v3.1.2)      | 44 min | 659,688  |      |           |
|               |           | uclust (v5.1.221)    | 46 min | 656,183  |      |           |
| TwinStudy 1M  | 1,000,000 | cd-hit-est (v4.6) 8 cores| 4 min  | 886,176  | 10   | 95%       |
|               |           | cd-hit-est (v4.6)    | 19 min | 886,176  |      |           |
|               |           | cd-hit-est (v3.1.2)  | 47 min | 893,044  |      |           |
|               |           | uclust (v5.1.221)    | 56 min | 890,549  |      |           |
| HumanGut 4M   | 4,000,000 | cd-hit-est (v4.6) 8 cores| 8 min  | 3,799,745| 10   | 95%       |
|               |           | cd-hit-est (v4.6)    | 42 min | 3,799,745|      |           |
|               |           | cd-hit-est (v3.1.2)  | 494 min| 3,811,372|      |           |
|               |           | uclust (v5.1.221)    | 214 min| 3,787,384|      |           |

1. All these programs take a fraction identity as the identity cutoff parameter, namely, 0.6 is used for 60% and 0.7 for 70% etc.;
2. CD-HIT 4.6 command line: program -i dataset -o output -c identity -n word -M 0 -T nthread. Number of threads nthread is 1 for tests using one core and 8 for 8 cores; For DNA sequence dataset, -r 0 is used to avoid checking reverse complements for sequence similarities;
3. CD-HIT 3.1.2 command line: program -i dataset -o output -c identity -n word -M max_memory. Maximum memory max_memory is 400 and 1000 (MB) for the SWISS-PROT and HumanGut datasets, and 5000 (MB) for the other two;
4. UCLUST 5.1.221 command line: uclust --cluster dataset --seedout output --id identity --w word --k sword --band 20 --maxlen 1000000 --minlen 10. Seed word length --k sword takes the default values (3 for amino acids and 4 for nucleotides).

Since there is no golden standard datasets for testing the biological accuracy of such sequence clustering programs, we use BLAST to evaluate the algorithmic accuracy of these programs. Because different programs use different heuristics to improve the speeds, some redundant sequences will be missed and clustered as representative (non-redundant) sequences, so the representatives produced by each of the programs may contain a different amount and small fraction of redundant sequences which can be identified by BLAST.

For each set of representatives, we define the redundancy ratio as the fraction of missed redundant sequences among the cluster representative sequences, and use it as a measure of program’s clustering accuracy. For example, after clustering by a program (e.g. CD-HIT 4) at certain identity (e.g. 60%), we run all-by-all BLAST (BLASTP or BLASTN) search on the
non-redundant sequence produced by CD-HIT 4 and identify redundant sequences at 60% by blast alignments, which are the missed redundant sequences by CD-HIT 4. The redundancy ratio is missed redundant sequences / total representative sequences. A smaller redundancy ratio indicates a better cluster quality. The redundancy ratios of the cluster representatives produced by the three programs are shown in the Table S2. Due to the computational time of all-by-all BLAST searching, only subsets (the first 50K) of the cluster representatives of the small datasets are evaluated.

Table S2: Redundancy ratios of cluster representatives

| Dataset    | CD-HIT 3 | CD-HIT 4 | UCLUST 5 |
|------------|----------|----------|----------|
| SWISS-PROT | 0.031    | 0.021    | 0.064    |
| TwinStudy  | 0.0026   | 0.0009   | 0.012    |

Table S3: Tests of the enhanced CD-HIT on the full-sized datasets

| Dataset    | #Sequence | Program     | #Core | Time       | #Cluster | Word | Identity |
|------------|-----------|-------------|-------|------------|----------|------|----------|
| NR         | 12,054,819| cd-hit (v4.6)| 1     | 1440 min   | 4,933,074| 5    | 70%      |
|            |           |             |       | 8          | 240 min  | 4,933,074|
|            |           |             |       | 24         | 176 min  | 4,933,074|
| TwinStudy  | 8,294,694 | cd-hit-est (v4.6)| 1     | 966 min   | 4,279,176| 10   | 95%      |
|            |           |             |       | 8          | 153 min  | 4,279,176|
|            |           |             |       | 24         | 89 min   | 4,279,176|
| HumanGut   | 23,285,083| cd-hit-est (v4.6)| 1     | 1464 min  | 17,613,444| 10   | 95%      |
|            |           |             |       | 8          | 237 min  | 17,613,444|
|            |           |             |       | 24         | 147 min  | 17,613,444|

1. The parameters used in this table are the same as those used in the Table S1 for CD-HIT 4.6, except that cd-hit-est uses option -r 1 to check the reverse complementaries for sequence similarities.

2.2 Parallelization Speedup and Scalability Against The Number of CPU Cores

To assess the effectiveness (speedup) of the proposed parallelization technique, we tested the new CD-HIT with different number (1, 2, 4, 8, 16, 24 and 32) of threads on the same set of datasets (full sized version). The run time of tests with 1, 8 and 24 threads on the three large full sized datasets are listed in Table S3. The results (Figure S3, left) demonstrate that the parallelization speedup scales well with up to 24 CPU cores, before it starts to degrade. The parallelization performs particularly well with 8 and less number of cores, in fact, for the big datasets, the speedups are almost linear and approach the number of used cores. Before the parallelization starts to degrade for large number of cores, it is also clear that higher speedup
is achieved on bigger datasets with few exceptions. There are a few factors that could have possibly contributed to such performance degradation, such as context switching for multiple threading, limited cache size and memory bandwidth etc. Since the speedup did not degrade on the smallest dataset (SWISS-PROT) for using 32 cores, the degradation is more likely due to the limited cache size and memory bandwidth. When the number of cores increases, the immutable shared word table will be accessed more frequently, which will result in more cache misses and more main memory accessing for large word tables. Main memory access is much slower than cache access due to latency and limited bandwidth, with increasing number of cores, at certain point, it will become a bottleneck for the speed. The tests on the SWISS-PROT dataset used smaller word size and a lower identity cutoff, which led to small word tables, so that the cache size and memory bandwidth limit had none or little impact on the speed when more cores are used.

Figure S3: Evaluation of the parallelized CD-HIT: parallelization scalability against the number CPU cores. Left: computational time speedup with respect to the number of used CPUs. Right: average CPU load as the percentage CPU usage per thread.

The effectiveness is also evaluated by the average load per CPU in the parallelization. The average load per CPU is computed as the ratio between the total CPU time of all used CPUs and the execution time multiplied by the number of CPUs. The results (Figure S3, right) indicate that the load per CPU is generally high and drops slightly when the number of CPUs increases. And as expected, the CPU load is generally higher for bigger datasets than for smaller datasets.

2.3 Parallelization Speedup and Scalability Against Dataset Sizes

To see how the parallelization scales with the size of datasets, we created subsets of the datasets with approximately 1, 2, 4, 6 and 8 million sequences, and then run CD-HIT on them without parallelization and with parallelization using 8 cores. Some of the tests have been run multiple times, but only small variations are observed in the speeds, so we only present the results of
one run per test. The speedups and average CPU loads are shown in Figure S4. The left figure shows the speedups increase with the sizes of input datasets increase, until the best speedups are achieved at around 6 million sequences. While for CPU loads, there is an apparent trend in which the load per CPU increases when the input dataset size increases.

Figure S4: Evaluation of the parallelized CD-HIT: parallelization scalability against dataset sizes. Left: computational time speedup with respect to the input dataset size. Right: average CPU load as the percentage CPU usage per thread.