Parallelization of Protein Clustering Algorithm Using OpenMP

Dhruv Dhar(✉), Lakshana Hegde, Mahesh S. Patil, and Satyadhyan Chickerur

Department of Computer Science and Engineering, B.V.B College of Engineering and Technology, Centre for High Performance Computing, KLE Technological University, Hubballi 580031, Karnataka, India
dhruvdharl@gmail.com, lakshanaghegde@gmail.com, mahesh_patil@bvb.edu, chickerursr@kletech.ac.in

Abstract. Proteins are the building blocks of all living organisms and its analysis can help us to understand the bimolecular mechanics of living organisms.

Protein clustering attempts to group similar protein sequences and has diverse applications in bioinformatics. However, this operation faces various computational challenges because of dependency on complex data structures, high memory usage and irregular memory access patterns. In genome studies, the time consideration for alignment is also an important parameter and should be minimized.

Conventional solutions have rather been unsuccessful in achieving decent runtime performance because these algorithms are designed for serial computation which means that they use a single processor to perform computations. These algorithms can be improved upon by modifying them to use multiple processing elements.

The purpose of this research is to modify existing protein clustering algorithm and apply parallelization techniques on them in order to optimize protein sequencing operation for faster results without sacrificing accuracy.

Keywords: Proteins · Protein clustering · Bioinformatics · Protein sequences Serial computation · Parallelization

1 Introduction

Clustering or cluster analysis is the process of grouping objects in such a way that the objects within a group have more similarities to each other than the objects in other groups. Each group is referred to as a cluster. Each cluster can have different size and the number of clusters that will be generated is not known at input. Clustering process can also be employed to find out the relationship between each cluster.

Clustering has numerous applications in the field of computational biology some of which include sequence analysis, clustering similar genes based on microarray data, gene expression analysis. In this paper, our focus will be on using cluster analysis for

M. Singh et al. (Eds.): ICACDS 2018, CCIS 906, pp. 108–118, 2018.
https://doi.org/10.1007/978-981-13-1813-9_11

grouping similar protein sequences. Our work is based on the serial *pclust* algorithm by Ananth et al.

Though clustering may seem to be a powerful algorithm for bioinformatics, its use is limited and it cannot be applied to all projects. This is because clustering is a data-intensive process and can easily become compute-intensive as well [3, 4].

The performance of the serial implementation of these algorithms is generally limited. These algorithms also face scalability issues. That is why the serial pclust algorithm does not scale beyond 15K–20K sequences on a desktop computer with 2 GB of RAM due to memory requirements [3, 4].

Parallelization techniques can be used to improve these algorithms. Parallelization can not only help in improving the run-time performance but can also help in achieving higher scalability with better results. We have tried to leverage multi-core computing architecture to solve the problem of protein clustering in parallel. In this project, we use OpenMP which is a shared memory parallelization library. OpenMP allows the programmer to explicitly create multiple threads. A thread is a basic unit of execution and can be scheduled parallelly onto multiple cores for simultaneous execution of multiple tasks. We have chosen OpenMP because it is easy to use compared to some conventional multi-threading libraries like POSIX and MPI.

While writing parallel programs, it should be made sure that all the threads are properly synchronized. Improper or incorrect synchronization may cause race condition leading to the generation of incorrect results. OpenMP provides various synchronization constructs like barrier, atomic and critical. However, it should also be noted that there is a certain amount of overhead associated with these constructs so the use of synchronization constructs must be minimized within the code.

This modified pclust algorithm which we have named as "*pclust-v7*" stands out from the conventional pclust algorithm not only because it provides better performance and output but also it offers better output visualization by use of bar graphs and pie charts. We have deployed our code in the cloud which helps us to achieve better security and flexibility of use. The software can be accessed from any client device at any location.

2 Literature Survey

With the evolution of high-performance workstations, parallel computing has attracted a lot of interest. In parallel computing, an application is designed in such a way that it can run on multiple processing elements simultaneously. For example, consider a for loop with 8 iterations and each iteration requires 1 unit of processing time. If we run the for loop on a single processor, the for loop will consume 8 units of time. Now consider a computing system with 4 processing elements. The for loop iterations are divided among the 4 processing elements so each processor gets 2 iterations to compute. If each of these processors perform their computations parallelly, the system would require only 2 units of time for computation leading to 4 times performance gain. Under the practical scenario, this is not the case as there are many overheads associated with parallel programs including synchronization overheads, idling, and load imbalances. It is the responsibility of the developer to minimize these overheads.

Our survey showed us that parallel computing is one of the best ways that can be used to optimize computation. Parallel computing has been employed in various areas of computational research for a long time. We tried different kinds of parallelization techniques on different algorithms before applying them on the pclust algorithm and noticed that parallelization significantly improves application performance for large input.

In spite of the numerous applications of clustering in computational biology, it is considered a dampening computational task due to involved complexities. In computational biology, it is also difficult to find suitable datasets.

There are two major classes of clustering methods which are hierarchical clustering and partitioning. In hierarchical clustering, each cluster is subdivided into smaller clusters, leading to a tree-shaped structure or a Dendrogram [15]. In partitioning method, the data is divided into a predetermined number of subsets where there is no hierarchical relationship between clusters [15]. The quality of clusters can be evaluated based on how compact and well separated the clusters are.

In biological areas, graph algorithms are widely used in biology network field such as drug target test, sequencing analysis, and alignment in getting to know the functions of various proteins and genes, to find the relationship between diseases and determining the antidote for them.

Biological research areas involve large computations involved in the field of molecular biology such as molecular modelling and developing an algorithm for analysis. Computations are also utilized by biogenetics, neural sciences etc.

As they involve a large number of computations and network analysis along with large datasets required for accurate results, it is better and more efficient to use parallel programming, as it would assist to reduce the time taken and often scales with the increase in the dataset. It also helps in making the program independent of the physical constraint of operating on a single processor (memory constraints etc.).

Clustering is one of the first steps carried out while performing gene expression analysis. This program focusses solely on clustering of proteins i.e. grouping similar proteins together. It uses shingling approach developed by Gibson et al. to perform clustering of protein molecules. This clustering algorithm can be used in various biological research fields. It is very important in the field of gene clustering where clustering similar genes are grouped to infer a function for each group. The clustering algorithm used in pclust can be used for gene clustering also. Optimizing the clustering process can help us to significantly reduce the time for performing expression analysis and other methods that involve biological clustering as a major step.

Before Pclust, BLAST algorithm was used universally for sequence alignment. In spite of its widespread use, BLAST cannot guarantee optimal alignment of sequences. The serial Pclust program makes use of shingling algorithm which occurs in two stages. In shingling algorithm, denser subgraphs are created if the vertices share s of their out links as such vertices are grouped together. As the value of s grows the probability that two vertices share the same shingle decreases. The algorithm develops c random shingles at the beginning for vertex v. As the value of c increases, the density of sub graphs also increases. Pclust works in three stages:

1. Shingling Phase I
2. Shingling Phase II
3. Connected Component Detection.

All these stages involve different types of computation but the basic parallelization techniques remain the same.

Several previous attempts have also been made in the same field. These have been discussed below:

- **Pclust-sm:** A parallel approach was developed by Ananth et al., for his OpenMP based implementation for clustering of biological graphs [3]. In his paper, he discusses use of hash tables instead of quick sort algorithm in order to reduce time complexity of the algorithm and thus reduce the overall runtime. Hash table is used to group together all the vertices generating a given shingle, thus eliminating the need for a separate sorting algorithm.
- **Pclust-mr:** We also came across a multistage MapReduce based implementation of serial graph clustering heuristic also developed by Ananth et al. [11]. The underlying algorithm transforms the Shingling heuristic operation into a combination of standard MapReduce primitives such as map, reduce and group/sort [11]. The algorithm was implemented and tested on a Hadoop cluster with 64 cores which did not perform very well.

3 Proposed Solution

A solution has been proposed to improve the performance of pclust protein clustering algorithm. This solution makes use of OpenMP library and involves the following steps:

1. Identifying the contention spots in the algorithm.
2. Determining how parallelization can be used to reduce or eliminate contention.
3. Applying OpenMP constructs to the algorithm.
4. Testing the parallelized algorithm for errors such as race condition and comparing its performance to the serial algorithm.
5. Verifying the results produced by the parallelized algorithm.

The algorithm involves a 2-pass Shingling process. The main idea of the Shingling algorithm is as follows: Intuitively, two vertices sharing a shingle. The algorithm seeks to group such vertices together and use them as building blocks for dense sub graphs [3, 4, 11, 16]. The input to the algorithm is a FASTA file with n sequences, variables s and c. Variables s and c denote the size of shingle and the number of trials respectively. Larger the value of s, lesser the probability that two vertices share a shingle. The parameter c is intended to create the opposite effect [3].

We start the parallelization process by modifying the init_vars function which is used to allocate memory to different variables. In the following code, allocation of one variable is completely independent from the allocation of other variables so rather than executing these statements serially, they can be run parallelly on different processors using the section construct. Consider the following code:

```
#pragma omp parallel
#pragma omp sections
  #pragma omp section
    vidmap = emalloc(gN*(sizeof *vidmap));
  #pragma omp section
    gA = emalloc(gC*(sizeof *gA));
  #pragma omp section
    gB = emalloc(gC*(sizeof *gB));
  #pragma omp section
    mfSglCnt = gN*gC;
    gFSgl = ecalloc(mfSglCnt, sizeof *gFSgl);
  #pragma omp section
    n2gidHash = ecalloc(gN, sizeof(*n2gidHash));
//end of sections
```

Next, we parallelize the free_vars function which is used to deallocate the variables. Here we are using the same approach as init_vars. However, instead of using separate sections for each free (memory deallocation) statement, we put four free statements inside one section. This will schedule four free statements to a single processor. We do this because the deallocation process is relatively less time taking. So if we schedule each free statement to a single processor, the overhead increases which is undesirable. Consider the following code:

```
#pragma omp parallel
  #pragma omp sections
    #pragma omp section
      free(vidmap);
      free(gA);
      free(gB);
      free_union(uSet);
    #pragma omp section
      free_sgl(gFSgl, fSglCnt);
      free_sgl(gSSgl, msSglCnt);
      free_hash();
      free_gid_hash(n2gidHash, gNN);
  //end of sections
```

We are only adding OpenMP constructs to the code and not modifying the logic of the algorithm until required. It is also important to note that some parts of the algorithm cannot be parallelized due to presence of I/O bound statements.

Parallelization can only be performed on CPU bound statements. For example, consider the function shingle which adds a lot to the total overhead due to presence of many for loops which are highly dependent on I/O.

It is very evident that for loops are the major contention spots in a program. Optimizing these loops can help to improve the run-time performance of the code. One

method of optimizing them can be by splitting the iterations and scheduling them on multiple processors.

Functions like free_hash(), free_gid_hash(), free_adjList(), free_sgl(), init_union(), init_vidmap() have for loops with CPU bound statements which can effectively be parallelized by using #pragma omp parallel for directive. Parallelizing these loops effectively reduce the time for which these loops run thus improving the overall performance. Consider the following for loops:

FREE_HASH ():

```
.....
  #pragma   omp   parallel   for schedule
  (dynamic,500) num_threads(4) shared(i)
    for(i=0; i<HASH_SIZE; i++)
      if(hashTbl[i])
        for(p=hashTbl[i]; p!=NULL; p=q)
          q= p->next;
          free(p->key);
          free(p);
  //End of pragma for
```

INIT_UNION (SIZE):

```
.....
  #pragma omp parallel for schedule(dynamic,150)
  shared(i) num_threads(2)
    for(i=0; i<size; i++)
      ufSet[i].parent = i;
      ufSet[i].rank = 0;
  //End of pragma for
return ufSet;
```

In both the code snippets, shared(i) has been used because variable i has to be shared among all the threads. Schedule(dynamic, n) means that n iterations will be dynamically allocated to any one of the available processors. Apart from the for construct, we also used constructs like task and other synchronization constructs like atomic and critical to making the algorithm more efficient and reliable.

A GUI interface was also created and attached with the algorithm for easy access to the algorithm. The GUI interfaces were created using Qt creator which produces '.ui' files as output. These '.ui' files were later converted to python files using piuic4 command. The graphs were created using python Matplotlib library. These python interfaces were attached to the c code. Following are some of the screenshots (Figs. 1, 2):

Figure 3 shows a bar graph which describes the number of members in each cluster having more than one member. This graph shows an overall trend that can be used to get quick insights.

```
ubuntu@ip-172-31-15-153:~/original_src_code/src_pclust$ ./kshingle -f tes
n 2230 -s 10 -c 1000
Initialization succeed...
#vertices:2230 => 1204
First level shingling succeed...
Second level shingling succeed...
Dumping clusters succeed...
time taken= 131.460232
```

Fig. 1. Shows the command line interface present in the original pclust algorithm. The command line arguments -f, -n, -s, -c denote the name of file, number of vertices, size of shingle and number of trials respectively.

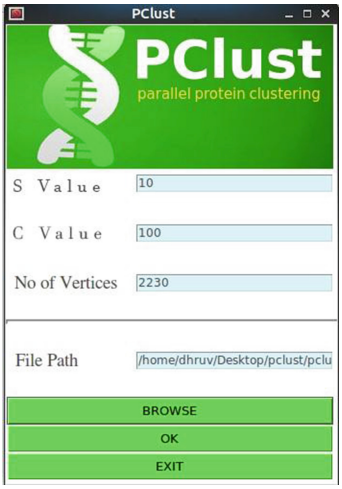

Fig. 2. Graphical user interface for the new program Pclust-v7.

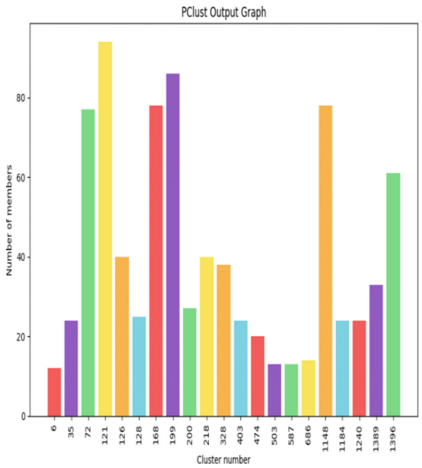

Fig. 3. Graph of cluster number v/s number of members

4 Results

In order to evaluate performance, both the serial and the parallel algorithms were deployed on the same machine and were run one after the other and the results were compared. The machine used by us had a 16 thread Intel Xeon-E5 2.3 GHz processor coupled with 32 GB memory. The dataset used was a FASTA file with 2230 protein sequences. The protein sequences look like following:

```
>ENS_PEP_ENSAPMP00000000218:>ENS_PEP_ENSCAFP00000
005470;
>JCVI_PEP_1096131504461:>JCVI_PEP_1096131655745;
>JCVI_PEP_1096133581735:>NCBI_PEP_27924029;
>JCVI_PEP_1096134202821:>ENS_PEP_ENSCAFP00000002867;
>NCBI_PEP_24582285:>ENS_PEP_ENSAPMP00000012063; etc…
```

Figure 5 shows the side by side runtime performance comparison between both the algorithms for s = 15. Note that we have randomly chosen s value as 15 but other values can also be used. We have kept the number of processing elements constant here (16). The blue bars denote the time taken by the serial pclust algorithm whereas the green bars denote the time taken by the parallel pclust-v7 algorithm.

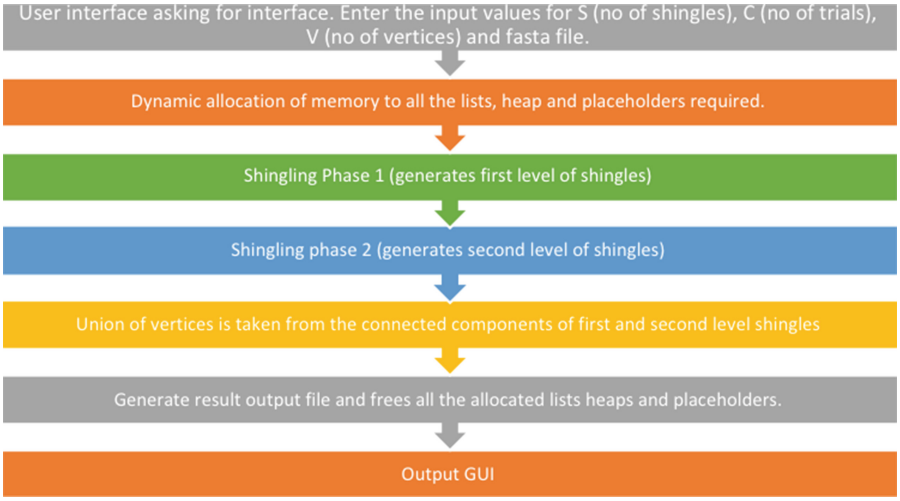

Fig. 4. Flow diagram listing all the processes involved in the algorithm.

We can infer from the graph that for large values of c, the performance gain is also higher. This happens because the total parallel overhead function, To is a function of both, problem size (W) and number of processing elements (p) used [1].

$$W = KT_o(W, p) \tag{1}$$

In many cases, the overhead increases sub-linearly with respect to the problem size. In such cases, the efficiency increases if the problem size is increased keeping the number of processing elements constant (in this case: 16). So the performance gain will continue to increase with increasing input size (Fig. 5). The following table shows the time taken for both the algorithms to complete clustering for various values of c (Table 1):

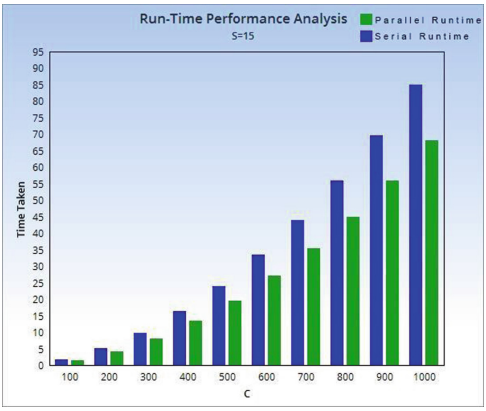

Fig. 5. Performance analysis of serial and parallel algorithms side by side for s = 15

The final output file displays clusters in the following format:

```
{Cluster#} 3
{Member#}  ENS_PEP_ENSCAFP00000029679
{Member#}  NCBI_PEP_33341450
{Member#}  NCBI_PEP_30354202
{Member#}  NCBI_PEP_28573069
.....
{Cluster#} 18
{Member#}  ENS_PEP_ENSDARP00000050238
{Member#}  NCBI_PEP_862457
{Member#}  NCBI_PEP_57098165
{Member#}  NCBI_PEP_55777830
.....
```

Table 1. The run-time (in seconds) of pclust and pclust-v7 on various input for s = 15. The variable *t* denotes the number of threads.

Number of trials	Pclust runtime (seconds)	Pclust-v7 runtime (s)	
		t = 4	t = 16
C = 200	6.23	5.89	4.6
C = 400	17.65	15.48	12.23
C = 600	33.84	30.87	27.56
C = 800	58.44	52.69	45.30
C = 1000	85.24	76.57	68.65

5 Conclusion and Future Scope

This paper describes a method to parallelize a protein clustering algorithm to make it more efficient. This algorithm performs better with large input as compared to the standard algorithm and also offers easy usage. The use of graphs also provides better output visualization. This algorithm is deployed on cloud, so hardware scaling can also be done flexibly when the need arises. The ability of pclust-v7 to cluster the proteins of hundreds of organisms on a desktop computer in a matter of minutes will allow scientists to conduct their research without the need to access expensive clustered computers.

Pclust algorithm has shown itself as a practical substitution for BLAST algorithm. In the future, we plan to extend the parallelization by use of libraries like CUDA which enables the algorithm to be executed on powerful GPUs instead of CPU. The scope of parallel computing is not just limited to bioinformatics but it can also be applied to other domains like Big Data, image processing, 3D-simulations, artificial intelligence etc.

The implementation discussed herein may not be highly precise and can still be improved further for higher accuracy.

Acknowledgements. The research was performed at Centre for High Performance Computing, KLE Technological University under the guidance of Prof. Mahesh S. Patil and Prof. Satyadhyan R Chickerur.

References

1. Grama, A.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Boston (2003)
2. Bioinformatics and Computational Biology Group, School of Electrical Engineering and Computer Science, Washington State University (2015–2016). Pclust Manual
3. Chapman, T., Kalyanaraman, A.: An OpenMP algorithm and implementation for clustering biological graphs. In: Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithm - IAAA 2011 (2011). https://doi.org/10.1145/2089142.2089146
4. Rytsareva, I., Chapman, T., Kalyanaraman, A.: Parallel algorithms for clustering biological graphs on distributed and shared memory architectures. Int. J. High Perform. Comput. Netw. **7**(4), 241 (2014). https://doi.org/10.1504/ijhpcn.2014.062724
5. Rytsareva, I., Kalyanaraman, A., Konwar, K., Hallam, S.J.: Scalable heuristics for clustering biological graphs. In: IEEE 3rd International Conference on Computational Advances in Bio and Medical Sciences (ICCABS) (2013). https://doi.org/10.1109/iccabs.2013.6629214
6. Introduction to OpenMP - Tim Mattson (Intel) [Video file] (n.d.). Accessed. https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG
7. OpenMP Architecture Review Board.: OpenMP Application Program Interface version 4.0. (2013). Accessed. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
8. Lockwood, S., Brayton, K.A., Broschat, S.L.: Comparative genomics reveals multiple pathways to mutualism for tick-borne pathogens. BMC Genom. **17**(1), 481 (2016). https://doi.org/10.1186/s12864-016-2744-9

9. Daily, J., Kalyanaraman, A., Krishnamoorthy, S., Vishnu, A.: A work stealing based approach for enabling scalable optimal sequence homology detection. J. Parallel Distrib. Comput. **79–80**, 132–142 (2015). https://doi.org/10.1016/j.jpdc.2014.08.009
10. Lu, H., Halappanavar, M., Kalyanaraman, A., Choudhury, S.: Parallel heuristics for scalable community detection. In: IEEE International Parallel and Distributed Processing Symposium Workshops (2014). https://doi.org/10.1109/ipdpsw.2014.155
11. Rytsareva, I., Kalyanaraman, A.: An efficient MapReduce algorithm for parallelizing large-scale graph clustering. In: Proceedings of the ParGraph' 2011 - Workshop on Parallel Algorithms and Software for Analysis of Massive Graphs, Held in Conjunction with HiPC 2011, Bengaluru, India (2011)
12. Computational Biology, 15 Jan 2015. Accessed. https://en.wikipedia.org/wiki/Computational_biology
13. Cluster analysis, 18 Jan 2018. Accessed. https://en.wikipedia.org/wiki/Cluster_analysis (2018)
14. D'haeseleer, P.: How does gene expression cluster work? Nat. Biotechnol. **23**, 1499–1501 (2006). https://doi.org/10.1038/nbt1205-1499
15. Gibson, D., Kumar, R., Tomkins, A.: Discovering large dense sub graphs in Massive graphs. In: Proceedings of the International Conference on Very Large Data Bases, pp. 721–732 (2005)