Improving Performance of Multiple Sequence Alignment Analysis in Multi-client Environments

Umit Catalyurek[†], Eric Stahlberg[+], Renato Ferreira[†], Tahsin Kurc[†], Joel Saltz[†]

† Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210
{catalyurek.1,ferreira.18,kurc.1,saltz.1}@osu.edu

+Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH, 43210
eas@osc.edu

Abstract

This paper is concerned with the efficient execution of multiple sequence alignment methods in a multiple client environment. Multiple sequence alignment (MSA) is a computationally expensive method, which is commonly used in computational and molecular biology. Large databases of protein and gene sequences are available to the scientific community. Oftentimes, these databases are accessed by multiple users to execute MSA queries. The data server has to handle multiple concurrent queries in such situations. We look at the effect of data caching on the performance of the data server. We describe an approach for caching intermediate results for reuse in subsequent or concurrent queries. We focus on progressive alignment-based strategies, in particular the CLUSTAL W algorithm. Our results for 350 sets of sequences show an average speedup of up to 2.5 is obtained by caching intermediate results. Our results also show that the cache-enabled CLUSTAL W program scales well on a SMP machine.

1 Introduction

An important goal of research in computational and molecular biology is to reach a better understanding of biological systems at the gene level. A more accurate knowledge of biological systems at this level carries a great potential for gaining insight into the causes and progression of many diseases such as cancer. A first step in the study of molecular biology is to analyze gene and protein sequences. When a new sequence is discovered, it is important to search sequence databases to find homologous sequences [1], and analyze structural and functional similarities and differences among multiple sequences [9, 11, 16].

A large number of research institutions, laboratories and biomedical companies are continually contributing sequence data to the rapidly growing databases of gene sequences. Many of these databases are publicly available for research and development, and provide a rich source of information for molecular studies. Examples of large gene databases include GenBank and PDB. For example, as of October 2001, GenBank of The National Center for Biotechnology Information (NCBI) contained about 13.5 million sequences with about 14 billion bases. Moreover, the size of the data has tripled in the last two years[1]. Thanks to faster networks, scientists can have online access to such databases from their desktop PCs or laptops. As a result, data servers for sequence databases should be designed to handle large number of queries submitted by many clients.

Multiple sequence alignment (MSA) analysis is a powerful means for analyzing structural and functional similarities and differences, and for finding historical and evolutionary relationships. MSA involves computing the alignment of three or more sequences and is a computationally expensive method. A large number of methods have been developed for fast and accurate multiple sequence alignments. Several research projects have focused on the development of heuristics [9, 11, 13, 16, 17], and the parallel implementations of MSA algorithms [12, 18].

Our work is concerned with the efficient execution of MSA methods in a multi-client environment. Client queries submitted against a database of sequences for MSA analysis usually involve comparison of known sequences (stored in the database) to one or more new sequences entered by the client. Moreover, in a multiple client environment, several clients may submit to a server requests that may contain overlapping subsets of sequences. In this paper, we look at the effect of data caching on the performance of the data server. We describe an approach for caching previously computed intermediate results. We focus on progressive alignment-based strategies, in particular the CLUSTAL W

[1]http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html

algorithm.

2 Progressive Alignment Algorithms

Progressive alignment-based strategies are the most commonly used methods for the solution of multiple sequence alignment problems, because of their efficient execution times [7]. The CLUSTAL algorithm proposed by Higgins [9] and its improved version, CLUSTAL W [17], are the most widely used methods among several alternatives. The basic idea behind progressive alignment is to generate an initial phylogenetic tree via a series of pairwise alignments and then, following the order in the phylogenetic tree, to incrementally build up the multiple sequence alignment. A progressive alignment-based method consists of three main stages: 1) computation of a distance matrix based on the alignments of all pairs of sequences, 2) calculation of a guide tree from the distance matrix, and 3) alignment of multiple sequences according to the branching in the guide tree.

Pairwise Alignment: The first step is the computation of a pairwise distance matrix. It is a symmetric, dense matrix, which contains a floating point value for each pair of sequences. The value represents the distance score of a sequence pair. The original CLUSTAL program uses a fast approximate method to compute the pairwise distance matrix. The CLUSTAL W program offers, in addition to the approximate method, a slower but more accurate solution using dynamic programming. For q sequences, the first step requires the computation of $q(q-1)/2$ pairwise alignments. As a result, the execution time complexity of this step is $O(q^2 l^2)$ for q sequences, each of which has an average length of l.

Computation of the Guide Tree: The second step is the calculation of a phylogenetic tree to guide the progressive alignment step. The tree is calculated from the distance matrix using the neighbor-joining method [15]. This iterative method successively selects the best sequence pair (i.e., the pair with the minimum distance score) to link until all pairs have been linked. After a pair has been linked, the distances of all other sequences to that pair are recomputed, and the corresponding values in the distance matrix are updated. The computational complexity of this step is $O(q^3)$.

Progressive Alignment: The last step is a constructive step, in which a series of pairwise alignments are computed using full dynamic programming to align larger and larger groups of sequences. The branching order in the guide tree determines the ordering of the sequence alignments. Starting from the leaves of the tree, the sequences are aligned toward the root. At each step either two sequences are aligned, or a new sequence is aligned with an existing alignment, or two existing alignments are aligned. The computational complexity of this step is $O(ql^2)$.

3 Sequence Analysis in a Multiple Client Environment

Many gene and protein databases can be accessed remotely over the Internet. In such a setting, a data server has to process multiple requests submitted by one or more clients. Queries associated with multiple sequence alignment analysis oftentimes involve comparisons of known sequences to one another. Even if a user submits a query involving a new sequence, the query generally contains multiple known sequences. If commonalities among queries can be utilized, better use of system resources can be achieved, and the overall performance of the data server can be improved.

Multi-query optimization is important in many application domains, such as relational databases [6, 14], deductive databases [5], decision support systems [19], and data analysis applications [2]. Several optimizations can be applied to speed up query execution when multi-query workloads are presented to the server. Multiple query optimization techniques mainly rely on caching common subexpressions. Carefully scheduling queries also plays an important role, because the execution of queries can be ordered in a way to better exploit expressions that have been already cached. In an earlier work [2, 3], we investigated optimizations for multiple query workloads in an application for visualizing digitized microscopy images. Our results show that significant performance improvements can be achieved by data caching and carefully scheduling queries.

In this section, we describe an approach for improving the execution of MSA analysis by caching previously computed results both in memory and on disk.

3.1 Data Caching

As was described in Section 2, a progressive alignment-based method consists of three main steps. We can view the output of a step as an intermediate result that is used by the next step. For example, the first step computes all pairwise alignments (and the associated scores) between the sequences in a query. The pairwise alignment scores are then used to produce a distance matrix, which is input to the second step that generates a guide tree. The guide tree is used to generate larger groups of aligned sequences progressively in the third step. Thus, the set of pairwise alignments, the distance matrix, the guide tree, and the groups of aligned sequences can be viewed as intermediate results. We argue that intermediate results computed by previous queries can be cached to speed up the execution of a new query.

In this work we have chosen to cache pairwise alignments computed in the first step so that they can be reused by other queries. This decision is based on two observations. First, the computational complexity of this step is the dominant factor in the total execution time. This theoretical analysis has been confirmed empirically by our experiments presented in Section 5, and by Mikhailov et. al. [12]. Hence, minimizing the execution time of this step can result in significant reduction in the overall execution time of a query. Second, the intermediate results generated in the last two steps largely depend on the relationships between the sequences in the query. Two queries should either be the same or have very similar characteristics so that the guide tree or the groups of aligned sequences can be reused. Each pairwise alignment, on the other hand, is computed separately and its contribution to the distance matrix is independent of other pairwise alignments. Therefore pairwise alignments generated by a query have greater potential for reuse by other queries than do the groups of multiple alignments and the guide tree.

When a query is received from a client, the cache is searched to find the sequence pairs that match the sequence pairs in the query– note that a query may contain new sequences as well as a subset of the sequences already in the database. The pairwise alignment step is performed only for the sequence pairs that are not in the cache. When a new pairwise alignment is computed, it is inserted into the cache. A challenging issue in designing the cache is to optimize the execution of search and insertion operations so that their overhead will not offset the performance improvement obtained from using cached results. The cache should be organized such that I/O overheads are minimized when accessing the portions of the cache that are stored on disk.

We propose a cache implementation based on B-Trees. We assume that each sequence in a database is assigned a unique identifier (e.g., a 32-bit number). A B-Tree is used to store pairwise alignments. A record in this B-Tree corresponds to a pairwise alignment, and stores the distance score of the pairwise alignment. The record is indexed by a key, which is obtained by concatenating the unique identifiers of the two sequences that make up the sequence pair. The smaller of the two sequence identifiers represents the upper half of a record key, whereas the larger one constitutes the lower half. In this way, only one copy of a sequence pair is needed to be stored in the B-Tree.

For each sequence in a query we should find the corresponding unique identifier, if it is not already provided by the query. Another B-Tree can be employed for this purpose. Sequences are used as keys in this B-Tree, and each record holds the unique identifier of a sequence. If a sequence is not in the B-Tree, it is assigned a unique identifier and inserted into the B-Tree. For example, if we use numbers for identifiers, we can maintain the current maximum identifier stored in the tree. A new sequence inserted into the tree is assigned the current maximum number plus one as its unique identifier. An alternative approach would be to use a hash table to store the unique identifiers for sequences. The hash table could be accessed using a hash function that computes an index into the hash table from a given sequence. However, the efficient implementation of the hash table requires a good hash function that will minimize collisions, and an efficient algorithm and data structure to resolve the collisions.

The cache structure described in this section effectively employs a two-tier B-Tree approach. The first tier B-Tree stores the unique identifiers of sequences, whereas the second tier B-Tree stores pairwise alignments. The size of the first tier B-Tree is linearly proportional to the number and length of sequences. However, the size of the second tier B-Tree grows with $O(q^2)$, where q is the number of sequences. Hence, using a single file for the second tier B-Tree may introduce technical and performance problems. First of all, some file systems (e.g., Linux `ext2`) limit the maximum file size to be less than 2GB. This places an upper bound on the number of pairwise alignments that can be cached, regardless of the size of available disk space. Moreover, a large, single file may incur a performance penalty, because of high I/O overheads (e.g., disk seeks) to search for a pairwise alignment in a large B-Tree. In order to alleviate these problems, the set of sequences can be partitioned into bins based on the unique identifiers of the sequences. The assignment of sequences to bins can be done in round-robin fashion or in blocks (i.e., a range of identifiers are assigned to the same bin). Each bin corresponds to a second tier B-Tree. The record for a sequence in the first tier B-Tree contains a pointer to the corresponding second tier B-Tree, in addition to the unique identifier of the sequence. A new pairwise alignment is inserted into the second tier B-Tree that is pointed to by the record, which corresponds to the sequence of the pair with smaller identifier, in the first tier B-Tree. In this way, only one copy of a pairwise alignment is stored in the cache. Note that if there is one-to-one correspondence between sequences and bins, a second tier B-Tree will contain the pairwise alignments for a single sequence and in this case only the second identifier can be used as a key.

With the organization of the cache as described above, on a distributed memory machine, the set of second tier B-Trees can be partitioned among the nodes in the system, thus allowing the effective use of aggregate storage space. The first tier B-Tree also can be partitioned across the system. However, we anticipate that in most cases that B-Tree will be able to be replicated on all the nodes, because the size of the first tier B-Tree will be small enough, as it is proportional to the number of sequences.

3.2 A Complexity Analysis of Proposed Approach

In this section, we analyze the computational complexity of the B-Tree based caching described earlier. We present the complexity analysis of the second tier B-Tree. A similar analysis can be carried out for the first tier B-Tree.

For complexity analysis, let us assume that the maximum number of keys in a tree node is given by $2t - 1$ and that the maximum number of pairs that can be stored in the cache is C. Then the worst case height of the tree is represented by $h \leq \log_t \frac{C+1}{2}$. This assumes a single second tier B-Tree and minimum number of keys per tree node. The CLUSTAL W algorithm that employs caching, starts its first step with $q(q-1)/2$ searches to the cache. The cost for a single search on a B-Tree is O($t \log_t C$), which leads to a total cost of O($q^2 t \log_t C$).

For a hit ratio of $H (< 1)$, each of the remaining $(1 - H)q(q-1)/2$ pairs should be computed. The cost of executing these computations is therefore equal to $(1-H)q(q-1)/2$O(l^2). Moreover, the cost of adding a new pair to the cache is also O($t \log_t C$). If another pair needs to be evicted from the cache, this extra cost may double, if the cost of finding a candidate for eviction is O(1). Now consider the I/O costs of maintaining and searching the cache. All three B-Tree operations (search, insert, and delete) require reading O($\log_t C$) tree nodes. However, if the first few levels of the tree can be pinned in memory, the number of disk I/O operations can be significantly reduced.

Let m be the number of misses, i.e. $m = (1 - H)q(q - 1)/2$, then the computational complexity of the proposed scheme is O($q^2 t \log_t C + ml^2 + mt \log_t C$). The number of blocks that need to be accessed is O($q^2(\log_t C - p) + m(\log_t C - p)$) where p is the number of B-Tree levels, which are pinned in memory.

3.3 Deployment on a SMP Machine

When the data server is deployed on a SMP machine, multiple queries can be executed simultaneously as separate processes or threads. The server instantiates multiple *query processes* (or threads)[2] to serve queries submitted by clients. A query received by the server is assigned to one of the query processes for execution. Care should be taken to make sure the integrity of the cache is maintained, as multiple processes may access the cache concurrently. In this section we discuss three different types of cache for servers running on SMP machines.

The first is a read-only cache. In this type of cache, all pairwise alignments that are likely to be accessed by many queries are precomputed and a cache is created from these pairwise alignments. When a query is executed, the pairwise alignments that are not in the cache are computed on the fly, but are not inserted into the cache. Although this method assures that the cache is uncorrupted, it requires good a priori knowledge of query access patterns to achieve good performance.

The second type is an online-updated cache. When a cache miss occurs, new data is inserted into the cache during query processing. Since multiple queries may submit updates to the cache at the same time, a locking mechanism should be implemented to serialize those updates. Moreover, duplicate updates by different queries should be eliminated to prevent the cache from growing unnecessarily. This cache type allows new pairwise alignments to be cached, so it does not require the knowledge of query access patterns. However, it may incur a high performance penalty because of the need to serialize accesses to the cache structure.

The third type is an offline-updated cache. This is a hybrid of the first two types. As for the first cache type, a cache is created by precomputing the pairwise alignments that are likely to be needed by queries. During query processing, each query process keeps a record of cache misses (i.e., the pairwise alignment and the associated distance score). The cache is updated when the system load is low– the cache misses that have been recorded by each query process are inserted into the cache. Note that cache misses recorded by a process can also be maintained in a local cache by that process. That is, each process is assigned a separate cache, which can be updated only by that process, and there exists a global cache, which is accessible by all processes. During query processing, no updates are allowed to the global cache. When a query is assigned to a query process, both the local cache and the global cache are searched to look for cached pairwise alignments. When a cache miss occurs, the query process updates the local cache only. Local caches are merged into the global cache when no queries are executing in the system or the system load is low. Note that locking on cache data structures is necessary only when the global cache is updated. Thus, this cache organization reduces the complexity and overhead associated with maintaining locks on cache data structures when there are multiple queries.

Updates to the global cache can be accelerated in several ways. Each processor can keep track of the number of times a pairwise alignment is requested. When merging local caches with the global cache, rarely used pairs can be eliminated. Moreover, if a two tier B-Tree structure is employed for the global cache, the global cache can be updated one second tier B-Tree at a time. A copy of the second tier B-tree that is in the process of being updated is created and this copy is updated using local caches. We need to lock a first tier B-Tree element, only when the new copy replaces the old copy of the second tier B-Tree and the corresponding pointer in the first tier B-Tree is updated. This scheme allows the queries that are currently executing in the system

[2] In general, the number of processes will be equal to the number of processors on the machine.

to access the global cache more efficiently.

4 CLUSTAL W Implementation with Data Caching

The CLUSTAL W program source code was minimally modified to support the introduction of primitive data cache operations search and insert. The delete operation was not needed for the investigation and was not introduced. The B-Tree code available from the GIST library [8] was utilized as the source of the data cache operations. A simple set of wrappers were written to bridge the C source code of CLUSTAL W to the C++ source code of the GIST library.

The modifications to the CLUSTAL W code were centered around enhancing the speed of the construction of the pairwise score matrix which is integral to the application. In this process, a matrix is created containing the pairwise alignment scores of each unique pair of sequences in the alignment. The data which was stored in the cache consisted of the names and lengths for both sequences, the gap weights and resulting pairwise score. The size of this data aggregate was only 40 bytes per element. The unique key for each pair combination in the data structure was a 20 byte composite key created by incorporating a mod of each sequence length into the name of each sequence and concatenating the result. It was necessary to include a length dependent property of the sequence into the key due to the lack of guaranteed uniqueness the sequence name alone provides.

Our current implementation employs a read-only cache for SMP machines. We are planning to implement the other cache types in near future.

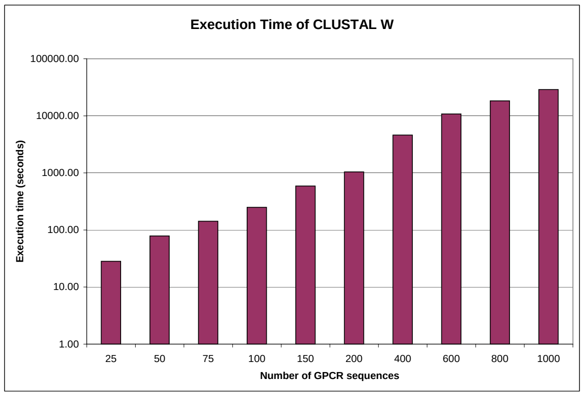
5 Experimental Results

In the first set of experiments, we performed a time profile analysis of the CLUSTAL W program (version 1.82). For these initial experiments, a PC running RedHat 7.1 Linux was used. The PC has one Pentium III 650MHz CPU, 768MB main memory and two 75GB IDE hard disks. We randomly selected 1000 sequences, with an average length of 450 amino acids per sequence, from a dataset of G-protein coupled receptor (GPCR [10]) protein sequences. We created 10 queries, with the number of sequences ranging from 25 to 1000 per query. Figure 1 shows the total execution time and the breakdown of the execution time into the three steps of the program, when the number of sequences in a query is varied. Note that the scale of y-axis (total execution time) in Figure 1(a) is logarithmic. As was discussed in the previous sections, the computational complexity of the pairwise alignment step for q sequences, with average sequence length of l, is equal to $O(q^2l^2)$, and is the dominant factor in the execution time of the CLUSTAL W program, as long as $q <= l$. Our experiments also confirm this theoretical result. As is seen from the figure, for queries with fewer than 400 sequences, the pairwise alignment step takes 78-95% of the total execution time. For example, with 400 sequences, CLUSTAL W ran 4606 seconds, where pairwise alignment step took 4342 seconds (94%). The execution time of the guide tree computation becomes more significant with the increasing number of sequences, after 600 sequences. The experimental results show that if pairwise alignment computations can be avoided, the overall performance can improve significantly.

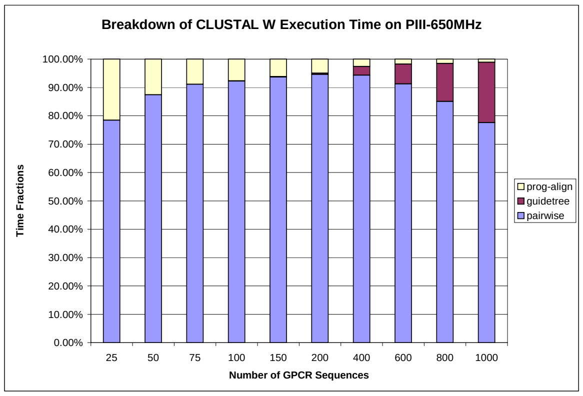
The rest of the experiments presented in this section were carried out on one of four Sun systems at the Ohio Supercomputer Center Sun Center of Excellence for High Performance Computing Environments. The model employed for these investigations was a 24-processor Sun Fire 6800 system, equipped with 750 MHz CPUs and 48 GB of main memory. The GNU compilers were used to compile both the CLUSTAL W code and the GIST library supporting the underlying cache data structure.

In the next set of experiments we examine the effect of cache on the performance of MSA. A full range of test cases were used in evaluating the modified CLUSTAL W algorithm. The length of the sequences ranged from 61 to 1580 characters, with an average value of 417. The number of sequences per alignment ranged from a low of 2 sequences in the alignment to an alignment involving in excess of 200 sequences. In all, over 350 sets of alignments were performed, with an average of 5700 characters involved in each alignment. The initial processing required computing the unique pair alignment entries and inserting them into the cache structure. Realizing that the number of elements to be computed and stored with even the 350 sequence set was too small to create a meaningful size dataset for benchmarks, an inflation factor was used during the loading phase of the cache. For each computed pair score introduced into the cache, 9 additional dummy entries were also inserted with a comparable but slightly modified combined key. This approach, while not a worst case situation, is far from ideal as it insures that each element of interest in the structure is surrounded by unutilized elements, thereby reducing any proximity benefit to be gained. The total number of elements inserted into the cache, which was used to perform the cache performance benchmarks in Figures 2 and 3, was in excess of one million individual entries, resulting in a file size in excess of 200MB.

Figure 2 shows breakdown of the execution time of MSA into the three steps, with and without data caching. Solaris, like many modern operating systems, maintains a system file cache. In order to emulate the case where the cache for multiple sequence alignment does not fit in memory, we employed the `directio` function in Solaris. The `directio` function is used to advise the OS to bypass the system file



(a)



(b)

Figure 1. The execution time of CLUSTAL W, as the number of sequences is varied. (a) Total execution time. (b) Breakdown of execution time into the three steps of the algorithm.

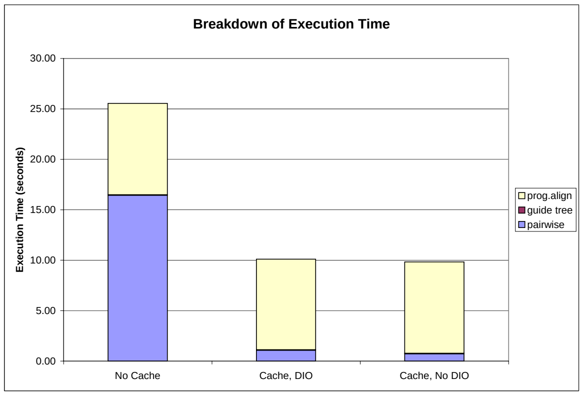


Figure 2. Breakdown of total execution time. In the graph, "Cache, DIO" and "Cache, no DIO" denote the cache-enabled version of CLUSTAL W with `directio` and without `directio`, respectively.

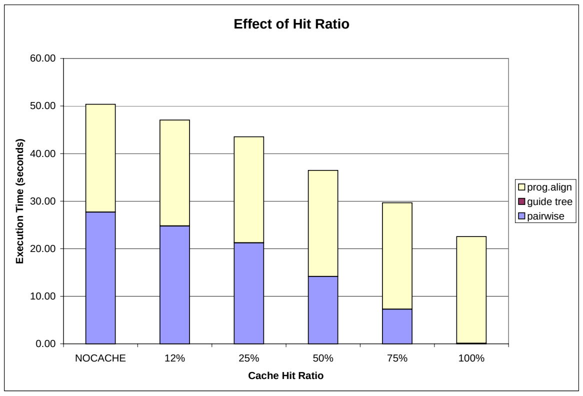


Figure 3. The effect on performance of the hit ratio.

cache for a specific file. As is seen from the figure, performance difference between the cache-enabled versions of CLUSTAL W with `directio` and without `directio` is very small and both of them performs significantly better than the original version of CLUSTAL W. Focusing on the pairwise score matrix created in the CLUSTAL W processing, the utilization of the cache approach provides a relative speed up that averages in excess of 15 over the 350 trial sets of sequences. In only one case was the cache recovery actually slower than computation, and this was for an alignment of 6 sequences which only required 3.1 milliseconds to compute. In this case, the retrieval of the elements required 3.3 milliseconds.

Figure 3 shows the average query execution time as the hit ratio is varied. As is seen from the figure, the cache-enabled version of CLUSTAL W performs better than the original version even with small hit ratio. As expected, the performance improves as the hit ratio increases. The execution time decreases almost linearly.

Figure 4 shows the total execution time as the number of query processes is varied. For this experiment, we created 64 disjoint sets of sequences. Each set had 40 sequences with unique identifiers. All the sequences in all the sets were of the same length. The pairwise alignments computed from each set were inserted into the cache used in the previous experiments, along with a set of dummy pairwise alignments, so that the final cache contained a total of 1.5 million entries. The size of the cache file was 400MB. In this experiment, we used a read-only cache, i.e., timing values do not include overhead of updating the cache. In the experiments, the same sets of sequences were also distributed among the query processes to carry out MSA. We assigned the same number of query sets to each query process so as to iso-

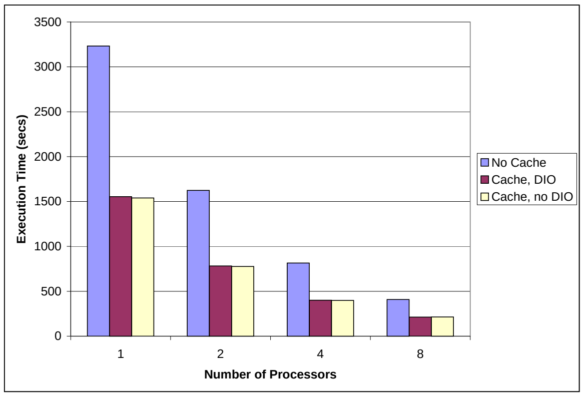


Figure 4. Total execution time of 64 queries, as the number of query processes is varied. In the graph, "Cache, DIO" and "Cache, no DIO" denote the cache-enabled version of CLUSTAL W with `directio` and without `directio`, respectively.

late the effect of caching on the scalability of the server. As is seen from the figure, when cache is disabled, the execution time of the queries decreases almost linearly. This is expected since the query sets have been evenly distributed among the query processes. When cache is enabled, the total execution time decreases to half for all configurations– note that the hit ratio in this experiment is 100%, since the same sets of sequences that have been used to create the cache are used as queries. Our results show that although all processes access the same B-Tree files concurrently, the associated overheads are small and the overall performance of the server scales well as the number of processes is increased.

6 Conclusions and Future Work

We have presented a caching approach for speeding up the execution of multiple sequence alignment queries in a multiple client environment. We have shown theoretical analysis indicating potential for substantial improvements on the overall execution time of the algorithms by using the caching schemes described in this paper. Experimental results have also been presented which corroborate our expectations. Overall, we were able to obtain an average speedup of 2.5 for the 350 trial sets of sequences. Our results also show that the cache-enabled CLUSTAL W scales well on a SMP machine.

One of our longer term goals is to evaluate different architecture configurations for MSA analysis and develop efficient support for alternative configurations. For example, it is becoming increasingly cost-effective to build large scale disk-based storage systems using currently available off-the-shelf components. A large disk-based storage system can be built at relatively low cost by connecting many such PCs via a high-speed switched network[3]. In addition to substantial storage space and high I/O bandwidth provided through the collective use of multiple disks in the system, such a configuration offers significant processing power for manipulating data, and large memory space that can be used for caching data. However, developing system support for MSA analysis on such systems is challenging. Several key questions should be addressed.

The first question is "which node should the query be executed on?". When a query is assigned to a processor, the corresponding pairwise alignments stored in the local caches of other processors should be communicated to that processor via inter-processor communication. The second question is "should a query reuse a cached pairwise alignment, or recompute it?". If a pairwise alignment is cached on a remote node and the network latency is high, the overhead of communication to retrieve the pairwise alignment from the remote node may be more than the cost of recomputing it. Good system support is needed to balance the workload among processors and minimize the communication overheads due to cached pairwise alignments. We plan to look at the implementation of system support using a component-based framework [4]. In this framework, data retrieval and data processing operations are implemented as a set of interacting components. Communication, computation, and I/O overheads can be minimized by placing components efficiently across the system and by efficiently scheduling data flow between components.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, Oct 1990.

[2] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient execution of multiple workloads in data analysis applications. In *Proceedings of the 2001 ACM/IEEE SC'01 Conference*, Denver, CO, Nov. 2001.

[3] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. Technical Report CS-TR-4290 and UMIACS-TR-2001-68, University of Maryland, Department of Computer Science and UMIACS, Oct. 2001. Submitted to IPDPS 2002.

[4] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[5] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of the 12th VLDB Conference*, pages 384–391, 1986.

[3] At Ohio State University, we are building a 7.2TB storage cluster with 24 PCs, each with 3 100GB disks, Pentium III 933MHz CPU and 512MB memory. At current prices, the cost of this cluster is about $50,000.

[6] F.-C. F. Chen and M. H. Dunham. Common subexpression processing in multiple-query processing. *Transactions on Knowledge and Data Engineering*, 10(3):493–499, 1998.

[7] D.-F. Feng and R. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J. Mol. Evol.*, 25:451–360, 1987.

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of 21st International Conference on Very Large Data Bases, Zurich*, pages 562–573, September 1995. Available at http://datasplash.cs.berkeley.edu:8000/gist/.

[9] D. Higgins and P. Sharp. CLUSTAL : a package for performing multiple sequence alignments on a microcomputer. *Gene*, 73:237–244, 1988.

[10] F. Horn, J. Weare, M. Beukers, S. Hrsch, A. Bairoch, W. Chen, . Edvardsen, F. Campagne, and G. Vriend. Gpcrdb: an information system for g protein-coupled receptors. *Nucleic Acids Res.*, 26(1):277–291, 1998.

[11] D. Lipman, S. Altschul, and J. Kececioglu. A tool for multiple sequence alignment. In *National Academy Science*, volume 86, pages 4412–4415, 1989.

[12] D. Mikhailov, H. Cofer, and R. Gomperts. Performance optimizations of Clustal W: Parallel Clustal W, HT Clustal, and MULTICLUSTAL. Technical report, SGI Life and Chemical Sciences, www.sgi.com/solutions/sciences/chembio, 1999.

[13] C. Notredame, D. Higgins, and J. Heringa. T-coffee: A novel method for multiple sequence alignments. *Journal of Molecular Biology*, 302:205–217, 2000.

[14] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM-SIGMOD Conference*, pages 249–260, 2000.

[15] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology Evolution*, 4:406–425, 1987.

[16] J. Stoye, V. Moulton, and A. Dress. Dca: An efficient implementation of the divide-and-conquer approach to simultaneous multiple sequence alignment. *Comput. Appl. Biosci.*, 13(6):625–626, 1997.

[17] J. Thompson, D. Higgins, T.J., and Gibson. Clustal w: improving the sensitivity of progressive multiple alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 222:4673–4690, 1994.

[18] T. Yap, O. Frieder, and R. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):1–12, 1998.

[19] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 271–282, Seattle, WA, 1998.