PARALLEL–TCOFFEE: A PARALLEL MULTIPLE SEQUENCE ALIGNER

Jaroslaw Zola
Iowa State University
Dept. ECE
Ames, IA 50011

Xiao Yang
Iowa State University
BCB Program
Ames, IA 50011

Adrian Rospondek
Czestochowa Univ. of Tech.
Inst. of Comp. & Inf. Sci.
42200 Czestochowa, Poland

Srinivas Aluru
Iowa State University
BCB Program
Ames, IA 50011

Abstract

In this paper we present a parallel implementation of *T–Coffee* — a widely used multiple sequence alignment package. Our software supports a majority of options provided by the sequential program, including the *3D–coffee* mode, and uses a message passing paradigm to distribute computations and memory. The main stages of *T–Coffee*, that is library generation and progressive alignment, have been parallelized and can be executed on distributed memory machines. Using our parallel software we report alignments of data sets consisting of hundreds of protein sequences, which is far beyond the capability of the sequential *T–Coffee* program.

1 INTRODUCTION

Multiple sequence alignment (MSA) is one of the most frequently performed tasks in computational biology. It appears as an essential component in areas such as database searching, identification of conserved motifs, and phylogenetic and phylogenomic inference, just to name a few [1, 2]. In general, MSA can be regarded as a method to capture similarity between protein or DNA/RNA sequences. MSA is a hard optimization problem for two main reasons: (i) it is very difficult to provide a formalization, e.g. alignment evaluation criterion, that would be satisfactory from a biological standpoint [3], and (ii) good modeling usually becomes very challenging algorithmically when the best (or optimal) alignment is desired [4]. Over the last few years several different approaches that address both these issues have been developed [5, 6, 7]: In the case of alignment evaluation, methods based on consistency [5, 7, 8] are considered to be the most biologically meaningful. As an efficient search strategy progressive alignment has been widely adopted [2, 9].

Consistency–based schemes, first introduced by Gotoh [10], exploit the observation that every MSA induces some level of consistency among pairwise alignments of its component sequences [5, 8]. By reversing this principle, pairwise alignments can be treated as guides or constraints on the MSA. Therefore, the MSA that agrees the most with all optimal pairwise alignments is believed to be the most accurate [8].

Progressive alignment [2] is by far the most popular alignment search heuristic. Strategies based on progressive alignment move toward the final solution step by step by following some predefined order (usually described by some binary rooted tree). At the beginning, the most similar pairs of sequences are aligned. Then, remaining sequences are added to the initial alignment, one by one.

The *T–Coffee* program [6] is one of the most successful approaches that combines consistency–based scoring function *COFFEE* [8] with the progressive alignment algorithm. The method has been shown to outperform other approaches in terms of alignment accuracy, and is widely adopted in bioinformatics [1].

1.1 T–Coffee

The *T–Coffee* (TC) algorithm works in two main phases: In the first phase TC constructs a list $\mathcal{L}$ of pairwise constraints, also termed the *library*. These constraints are subsequently used in the second phase to evaluate partial MSAs. Each constraint is a 3–tuple $\{s_{ix}, s_{jy}, w\}$, where s_{ix} denotes residue x of sequence i, there is some pairwise alignment or other evidence supporting the alignment of s_{ix} with s_{jy}, and w is the weight of the constraint, initially equal to the percent sequence identity of s_i and s_j. The library can be generated using different sources of information, and this makes TC a very flexible solution (in many cases superior to other MSA packages). The original method uses global pairwise alignment and ten best local non–intersecting pairwise alignments. More recently the *3D–coffee* mode has been introduced [11], which allows structural information about aligned proteins to be included in the library. In addition, the library can include data generated by other MSA software. Due to the heterogeneity of data sources, some of the constraints may appear in the library several times. To eliminate this duplication, residue pairs that occur multiple times are merged into one tuple with combined sum of the weights of individual occurrences. Finally, the library can be extended using transitivity property to include indirect information about constraints. For instance, if $\{s_{ix}, s_{jy}, w_1\} \in \mathcal{L}$ and

$\{s_{jy}, s_{kz}, w_2\} \in \mathcal{L}$ the constraint $\{s_{ix}, s_{kz}, w_3\}$ will be added to the library, where w_3 is a function of w_1 and w_2, even if there is no direct evidence for such a constraint in the source pairwise alignments. In the current implementation of TC, other extension schemes are provided as well (see *"T–Coffee Reference Manual"* for more details).

In the second phase, TC progressively aligns input sequences using information from the library. In this stage the library is represented as a three–dimensional lookup table: for each sequence and every residue a list of associated constraints is stored. The order of alignment is determined by the binary guide tree which is a *neighbor–joining* tree [12]. Leaves of the tree are input sequences, internal nodes correspond to the partial MSA, and the final solution is stored in the root node. To find the optimal MSA in a given node, which corresponds to the alignment of all sequences at the leaves in the subtree of the nodes, standard dynamic programming [13] can be used (however, TC provides a few other methods). Suppose that two alignments, A_L and A_R, are to be merged together. To compute entry $[x, y]$ of the corresponding dynamic programming matrix all constraints that are related to column x in alignment A_L and column y in alignment A_R are combined together. In that way, even in the early stages of the progressive alignment, a complete information about pairwise alignments is taken into account. This avoids errors typical of the classic progressive alignment approach.

TC is one of the most accurate MSA methods available. Unfortunately, its high accuracy is achieved at the expense of memory and time complexity. As already mentioned, TC uses pairwise alignments as the main source of constraints. Therefore, to generate the library for n sequences, $\binom{n}{2}$ global and local alignments have to be computed. Moreover, in the *3D–coffee* mode, additional $n \cdot m$ sequence–structure and $\binom{m}{2}$ structure–structure comparisons have to be performed, where m is the number of available protein structures (usually $m \ll n$). The number of constraints that have to be stored in the memory (size of the library) is proportional to $n^2 \cdot l$, where l is the average length of the input sequences. Finally, the progressive alignment requires $n - 1$ partial multiple alignments to be performed, and because at this stage the library information is used, each alignment can be compute intensive. As a result TC does not scale beyond more than 100 sequences, which limits its potential applications.

In this paper we present *Parallel T–Coffee*, the first parallel implementation of the *T–Coffee* method. Using distributed memory machines our software can align hundreds of sequences within reasonable time limits. To achieve this, PTC distributes the constraints library among computational nodes, and performs alignment operations in parallel using dynamic scheduling techniques.

2 PARALLEL T–COFFEE

We based our parallel implementation on *T–Coffee 3.79*, the most recent version of TC available at the beginning of our project. In this release TC provides a rich user interface and supports several different methods for pairwise alignment and the constraints library extension. It also implements the *3D–coffee* mode, and allows for remote communication with RCSB, the protein data bank server [14]. To preserve the original functionality of TC we utilized most of its source code, reimplementing and improving only as needed for efficient parallel execution.

Our implementation uses a distributed master–worker architecture and the message passing paradigm. To implement distributed memory mechanisms, we have employed one–sided communication primitives offered by the MPI–2 standard.

2.1 Initialization

The execution of *Parallel TCoffee* (PTC) starts with parsing of the input sequences. If the *3D–coffee* mode is used, PTC will contact the RCSB server to download PDB files storing structures that match the input sequences. This process requires data transfer over the FTP protocol for every protein structure that matches one of the input sequences. To hide overhead caused by Internet communication we implemented a multithreaded prefetching mechanism: as soon as the identifiers of the input sequences are known, up to 8 threads are started on the master node and connect to the RCSB server. At the same time the main thread proceeds with all other initialization tasks. Once initialization is completed and all requested structures are downloaded, the master node distributes complete input data among all workers. Replication of these data is an obvious step as it will be required throughout the execution and only occupies modest memory.

As already explained, TC employs a progressive alignment search strategy guided by the neighbor–joining tree. To generate such a tree some measure of sequence similarity, expressed in the form of a distance matrix, is required [12]. Please note that exactly the same measure is used to weight elements in the constraints library. To render the distance matrix TC provides several methods that require $\binom{n}{2}$ "quick" sequence comparisons (e.g. based on k–mer counting) that in general do not require alignment reconstruction. Because computation of the distance matrix consists of totally independent tasks it is easy to parallelize: To distribute computations we use *Guided Self Scheduling* (GSS) [15], which is a well known strategy based on a simple management of a list of completed tasks. Each worker computes part of the distance matrix and it measures the time such computations require. This information is later sent to the master node so that workers can be ordered accordingly. As a result, dynamic scheduling in subsequent

stages can be improved. This may be important when PTC is executed in a heterogeneous environment.

2.2 Library Generation

Library generation is the most time and memory consuming part of TC, limiting its applicability to no more than 100 sequences on a typical workstation in practice. In addition, cost of the library generation is, in most cases, dominant in the total execution time of TC as it is quadratic in the number of input sequences. To overcome these limitations in PTC, both memory and computations required by the constraint library are distributed among workers.

The library generation proceeds in three phases. First, all pairwise constraints are generated. Next, they are grouped and reweighted as described in the Introduction. The library extension is not performed in this stage, but it is postponed to the progressive alignment step when it is done "on the fly" (see *"T–Coffee Reference Manual"*). Finally, the library is transformed into the three–dimensional lookup table which is utilized during progressive alignment.

The first step is similar to computing the distance matrix. The differences are that each pair of sequences is compared using at least two different methods (i.e., global and pairwise alignment), and alignment is always reconstructed to allow its list of constraints to be rendered. To perform this step we distribute all pairwise alignment computations using modified GSS. At this stage the approximated efficiency of the processors is known and it can be used to minimize the number of message exchanges between the master and workers. Specifically, half of the total number of required pairwise alignments is distributed proportionally based on worker priorities. Then, the other part is distributed using GSS. When a worker completes its part of the computations it generates a list of the corresponding constraints and stores it in its local memory.

The next step is to eliminate duplicate entries in the distributed constraint list. To achieve this we take advantage of the fact that merging partial constraint lists is simply the accumulation of weights and therefore is associative. First, each host merges constraints locally using original TC algorithm. Then, we use parallel sorting to group repeat constraints that are found in different workers. A constraint $\{s_{ix}, s_{jy}, w\}$ is assigned to the bucket $b = (i + j) \ mod \ p$, where p is the number of processors including master. Next, each processor applies the sequential merging to the constraints in the bucket it stores.

In the last step the library is turned into a lookup table. Rows of the table are indexed by sequences and columns are indexed by residues. Element $[x, y]$ of the table stores all constraints $\{s_{xy}, s_{ij}, w\}$. As a result the table guarantees a unit cost access to the constraints that are bound to a given residue in a given sequence. As already mentioned in the Introduction, these data are required to construct the

dynamic programming matrix whenever two partial alignments are combined. Suppose that alignment A_L of sequences $\{s_a, s_b, s_c\}$ is to be aligned with the alignment A_R of sequences $\{s_d, s_e\}$. To evaluate entry $[x, y]$ of the corresponding dynamic programming matrix all constraints that are related to the residues $s_{(a..c)x}$ and $s_{(d..e)y}$ are required. This implies that the part of the lookup table indexed by the sequences $\{s_a, \ldots, s_e\}$ will be accessed. Note that in the last stage of the progressive alignment the entire lookup table will be accessed. Finally, since TC offers various alignment algorithms and we want to parallelize the progressive alignment stage it is impossible to identify exact pattern of lookup table requests. To satisfy requirements coming from the above we have implemented lookup table using one–sided remote memory access mechanisms (RMA) and caching techniques similar to [16].

In our approach the lookup table is divided row–wise such that every processor manages n/p rows. All rows assigned to a given processor are stored in an array, and two accompanying indexing vectors, one storing row offsets and one storing column offsets, are created. These vectors are exchanged among all workers, so that each worker can easily retrieve exact address of any entry in the lookup table. Next, each host creates a read–only RMA window that exposes its part of the lookup table to other processors. To access a remote part of the lookup table, the processor that manages the requested row is identified and the entire row is prefetched to local memory. As a result we may expect that all other requests related to this row (thus, to the particular sequence) will be served locally. Note that operations performed on the lookup table are read only, and therefore do not require additional mechanism to maintain coherence. Moreover, one–sided communication primitives can benefit from architecture dependent solutions (e.g., RDMA in the case of InfiniBand network [17]).

To handle frequent requests to remote memory, and to store prefetched data locally, we have implemented a flexible caching system. Every prefetched part of the lookup table is put into the cache whose size is specified by the user. The cache is managed using LRU policy, which we found to perform the best as compared to other policies, e.g., GDS, LFU or Min–Size. Application of the caching greatly reduces communication rendered by remote read operations. It implies also partial and self–adaptable replication of the lookup table. This property will be very useful during the progressive alignment stage.

2.3 Progressive Alignment

Progressive alignment is the last and the most difficult to parallelize step of the TC algorithm. Computations in this stage follow a tree order, thus parallelization can be reduced to directed acyclic graph (DAG) scheduling problem. Precedence constraints imposed by the binary tree cause that in the optimal case (i.e., when the tree is perfectly bal-

anced) the maximal speedup is bounded by $n/\log(n)$. Note that this is a rough estimate that assumes that tasks have similar size. At the same time computations that correspond to a single node in the graph require complete output from the preceding tasks. Finally, the constraint library is distributed which means that some of the tasks will require remote memory access. At this point we should mention that unlike other progressive alignment methods, e.g. *ClustalW*, TC spends significant amount of time in the progressive alignment stage.

To schedule progressive alignment tasks we have decided to use a strategy similar to the HLFET (Highest Level First with Estimated Times) algorithm [18]. Our algorithm schedules a graph node to a processor that allows the earliest start time, and graph nodes are ordered with respect to estimated execution time. Priorities of tasks and list of workers are updated every time one task is completed and another one is to be scheduled.

Consider graph $G = (V, E)$, where V is a set of internal nodes (tasks) of the progressive alignment guide tree, $|V| = n - 1$, and E is a set of edges describing precedence constraints between tasks. Edge e_{ij} is introduced if alignment corresponding to node v_i depends on the alignment corresponding to node v_j. We denote by $A_L(v)$ and $A_R(v)$ multiple sequence alignment associated with left and right predecessor of v, respectively. We denote size of the alignment (i.e., number of its component sequences) by $|A|$ and its length by $\overline{A}$. We describe by $L(v)$ and $R(v)$ number of internal nodes that have not been processed in the left and right subtree of v, respectively. Finally, we define $T(v) = L(v) + R(v)$ and $S(v) = \overline{A_L(v)} \cdot \overline{A_R(v)} \cdot |A_L(v)| \cdot |A_R(v)|$. To schedule graph G we use following criteria:

- Node v_i has a higher priority than node v_j if $T(v_i) < T(v_j)$,

- If $T(v_i) = T(v_j) = 0$, node v_i is scheduled before v_j if $S(v_i) < S(v_j)$.

The first criterion allows to separate nodes that are ready to schedule from nodes that are constrained by unfinished preceding tasks. The second criterion is the actual condition deciding about scheduling order. Here $T(v)$ is an approximation of the time required to execute task v: To generate MSA associated with node v, a dynamic programming matrix of size $\overline{A_L(v)} \cdot \overline{A_R(v)}$ is constructed, and to compute score in a single cell of the matrix around $C \cdot |A_L(v)| \cdot |A_R(v)|$ constraints have to be examined, where $C > 0$ is a constant. In our approach we neglect the time to communicate between scheduler and processors (which can be captured as weights of the edges in the graph G) as it is very small compared to the computation time.

The above criteria give higher priority to the tasks that are unbalanced, i.e., $0 \ll \big||A_L(v)| - |A_R(v)|\big|$, and have short execution time. As a result we may expect that scheduling delays due to precedence constraints will be minimized.

In addition to the task list we manage a list of workers which describes order of task assignment. Processors in the front of the list are assigned tasks with the highest priority. In the first few steps of the algorithm (i.e., when the number of tasks ready to schedule is greater than the number of processors) we prioritize hosts using performance measures obtained during distance matrix generation. As soon as the number of available processors becomes greater than the number of tasks to schedule we use another criterion, which is size of the library cached by a given worker. Each worker attaches a "piggyback message" with size of the cached data when sending request to the master for task assignment. This information is used by the master in the following way: If a task to be scheduled depends on the task that has been processed by the requesting worker it is assigned to this worker. Otherwise, the task is assigned to the worker with the largest data cached. In this way we try to maximize locality of references in accessing the library. Obviously, this has a significant impact on the number of messages generated by remote memory access.

3 PERFORMANCE EVALUATION

To validate our software we have performed a set of experiments with protein data from the Pfam database [19]. The experiments were conducted on our cluster consisting of dual Intel Xeon 3GHz nodes. Each node is equipped with 2GB of RAM, and runs Linux. The cluster is connected by a FastEthernet network. In our experiments we utilized the *mpich2-1.0.4p1*, a well known and efficient implementation of the MPI standard. Our software was compiled with the *GCC-3.4.4* set of compilers.

3.1 Experiments

Using *Parallel T–Coffee* we have aligned three data sets: PF00074, 349 sequences with maximal length 140 amino acids (AA); PF00231, 554 sequences, maximal length 331 AA; PF00500, 1048 sequences, maximal length 523 AA. These data sets are too large to be processed by the sequential *T–Coffee* program due to both memory and computational requirements.

The experiments have been run on 16 to 80 CPUs. In each experiment, every processor could use 768MB of main memory as a cache storage. Results of the experiments (averaged over 8 executions for each case) are summarized in Table 1 and Figure 1.

3.2 Discussion

As expected, execution time of the *Parallel TCoffee* decreases with the number of processors. Figure 1 shows that relative speedup (computed with respect to the execution time for 16 CPU) of the constraints library generation

Table 1: Running time of Parallel T–Coffee (in seconds).

CPU	PF00074	PF00231	PF00500
16	702 (558)	6014 (4000)	28495 (22488)
24	493 (366)	4379 (2615)	20329 (14756)
32	402 (273)	3704 (1962)	16432 (11012)
48	313 (182)	2804 (1295)	12386 (7296)
64	246 (137)	2405 (970)	10392 (5460)
80	243 (110)	2398 (775)	9264 (4368)

Time for constraint library generation is given in parenthesis.

Figure 1: Relative speedup of *Parallel TCoffee*.

is linear and independent of the size of input data. This is not surprising in light of the inherent parallelism in this stage.

Significantly different results (although not surprising) can be observed in the case of progressive alignment. To be cost optimal this stage requires that no more than $p = n/\log(n)$ processors will be used and the guide tree will be perfectly balanced. The first requirement is in opposition to the previous observations regarding constraint library generation (and note that in general this stage is dominant). The second condition is hardly ever satisfied for obvious reasons.

To better understand results presented in Figure 1 let us define a tree imbalance I_s as the number of internal nodes $v \in V$ such that $|A_L(v)| = 1$ or $|A_R(v)| = 1$ but not both at the same time. I_s can be normalized by dividing by $n - 2$. In this case $I_s = 0$ denotes balanced tree and $I_s = 1$ corresponds to completely imbalanced one. This measure differs from the classic Colless's coefficient I_m [20] as it captures only those nodes that constrain progressive alignment, thus we may consider it as a measure of difficulty of progressive alignment. The I_m coefficient on the other hand measures entire balance of the tree and can be combined with I_s as follows. Small values of I_s and I_m indicate that the tree is very well balanced and easy to schedule. If I_s grows and I_m is small, tree is imbalanced close to leaves and well balanced close to the root. Finally, if I_s is small and I_m grows, tree is imbalanced close to the root and balanced close to the leaves.

We have measured I_s and I_m coefficients for guide trees generated by PTC for our test data sets: PF00074 $I_m = 0.06, I_s = 0.39$; PF00231 $I_m = 0.03, I_s = 0.41$ and PF00500 $I_m = 0.07, I_s = 0.45$. As can be seen all three trees are rather unbalanced close to leaves and well balanced close to root. This situation is very undesirable because (i) tasks that are well balanced cannot be scheduled due to constraints in preceding subtrees, and (ii) good balancing of tasks close to the root causes that sequential part of the progressive alignment increases. Recall that execution time of task v is bounded by $|A_L(v)| \cdot |A_R(v)|$.

There are several alternative approaches that can be used to overcome current limitations of the progressive alignment stage. To generate partial multiple sequence alignment we use dynamic programming approach simi-
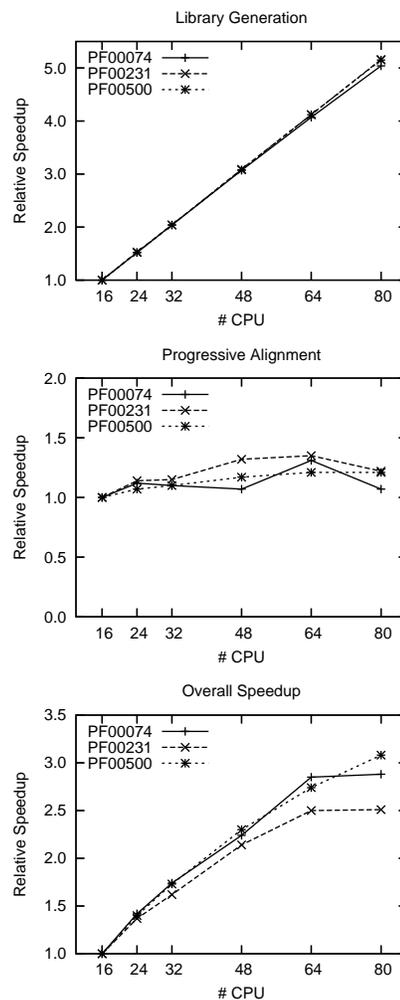
lar to the classic pairwise alignment. Therefore we can use parallel programming, e.g. approach proposed in [21], on the level of task and not tree. The main problem with this approach is that part of the constraint library required by given task has to be replicated on every node. This turns out to be impractical in the last steps of the progressive alignment. Another possible solution is to overcome precedence constraints by using partial output of the alignment to start successive tasks that otherwise could not be started. Unfortunately, this approach is very hard in practice due to the way TC implements dynamic programming. Finally, to utilize processors that are idle during progressive alignment it is possible to extend current TC algorithm with iterative alignment similar to [16, 22]. As a result we could expect significant improvement in the quality of generated alignments as several different guide trees could be investigated in a single run. This approach can be very easily integrated with current PTC framework.

Despite discouraging results of parallel progressive

alignment Figure 1 shows that overall speedup of PTC is reasonable. We believe that *Parallel TCoffee* can be valuable tool when large data sets have to be analyzed with precision that can not be guaranteed by other methods.

4 CONCLUSION

We have presented parallel implementation of *T–Coffee*, which is a popular multiple sequence alignment tool. We proposed several directions in which PTC can be improved. The purpose of our project is to provide stable and efficient implementation of the *T–Coffee* that would preserve the functionality of the original software and overcome its main limitations. Our implementation has been tested on all major platforms including 64–bit systems and is freely available together with documentation at the following URL: `http://www.ece.iastate.edu/~zola/ptc`.

REFERENCES

[1] Edgar, R., Batzoglou, S.: Multiple sequence alignment. Current Opinion in Structural Biology **16** (2006) 368–373

[2] Notredame, C.: Recent progress in multiple sequence alignment: A survey. Pharmacogenomics **3**(1) (2002) 131–144

[3] Thompson, J., et al.: Towards a reliable objective function for multiple sequence alignments. J. Mol. Biol. **314**(4) (2001) 937–951

[4] Elias, I.: Settling the intractability of multiple alignment. In: Proc. of ISAAC 2003. Volume 2906 of LNCS. (2003) 352–363

[5] Do, C., et al.: ProbCons: Probabilistic consistency–based multiple sequence alignment. Genome Res. **15**(2) (2005) 330–340

[6] Notredame, C., Higgins, D., Heringa, J.: T–Coffee: A novel method for fast and accurate multiple sequence alignment. J. Mol. Biol. **302**(1) (2000) 205–217

[7] Schwartz, A., Pachter, L.: Multiple alignment by sequence annealing. In: Proc. of ECCB 2006. (2006)

[8] Notredame, C., Holm, L., Higgins, D.: COFFEE: an objective function for multiple sequence alignments. Bioinformatics **14**(5) (1998) 407–422

[9] Chenna, R., et al.: Multiple sequence alignment with the Clustal series of programs. Nuc. Acids Res. **31**(13) (2003) 3497–3500

[10] Gotoh, O.: Consistency of optimal sequence alignments. Bull. Math. Biol. **52**(4) (1990) 509–525

[11] O'Sullivan, O., et al.: 3DCoffee: Combining protein sequences and structures within multiple sequence alignments. J. Mol. Biol. **340**(2) (2004) 385–395

[12] Saitou, N., Nei, M.: The neighbor–joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol. **4**(4) (1987) 406–425

[13] Gotoh, O.: An improved algorithm for matching biological sequences. J. Mol. Biol. **162**(3) (1982) 705–708

[14] RCSB: Protein data bank. http://www.rcsb.org/ (2007) (last visited).

[15] Polychronopoulos, C., Kuck, D.: Guided self–scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Trans. on Computers **36**(12) (1997) 1425–1439

[16] Parmentier, G., Trystram, D., Zola, J.: Large scale multiple sequence alignment with simultaneous phylogeny inference. J. Par. Dist. Comp. **66**(12) (2006) 1534–1545

[17] Jiang, W., et al.: High performance MPI–2 one–sided communication over InfiniBand. In: Proc. of IEEE CCGrid 2004. (2004) 531–537

[18] Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. J. Par. Dist. Comp. **59**(3) (1999) 381–422

[19] Bateman, A., et al.: The Pfam protein families database. Nuc. Acids Res. **32** (2004) D138–D141

[20] Colles, D.: Review of Phylogenetics: the theory and practice of phylogenetic systematics. Systematic Zoology **31**(1) (1982) 100–104

[21] Aluru, S., Futamura, N., Mehrotra, K.: Parallel biological sequence comparison using prefix computations. J. Par. Dist. Comp. **63** (2003) 264–272

[22] Wallace, I., O'Sullivan, O., Higgins, D.: Evaluation of iterative alignment algorithms for multiple alignment. Bioinformatics **21**(8) (2005) 1408–1414