Parallel implementation and performance characterization of MUSCLE

Xi Deng[1], Eric Li[2], Jiulong Shan[2], Wenguang Chen[1]

[1]Tsinghua University
Dept. of Computer Science
Beijing, 100084 China
dengx03@mails.tsinghua.edu.cn
cwg@tsinghua.edu.cn

[2]Intel China Research Center Ltd.
9/F, Raycom Infotech Park A, Zhong Guan Cun
Beijing, 100080 China
eric.q.li@intel.com
jiulong.shan@intel.com

Abstract

Multiple sequence alignment is a fundamental and very computationally intensive task in molecular biology. MUSCLE, a new algorithm for creating multiple alignments of protein sequences, achieves a highest rank in accuracy and the fastest speed compared to ClustalW as well as T-Coffee, some widely used tools in multiple sequence alignment. To further accelerate the computations, we present the parallel implementation of MUSCLE in this paper. It is decomposed into several independent modules, which are parallelized with different OpenMP paradigms. We also conduct detailed performance characterization on symmetric multiple processor systems. The experiments show that MUSCLE scales well with the increase of processors, and achieves up to 15.x speedup on 16-way shared memory multiple processor system.

1. Introduction

Multiple sequence alignment is a fundamental and challenging problem in computational molecular biology [1, 3, 11]. It can be used to find conserved regions in biomolecular sequences, predict the protein structure, and help constructing phylogenetic tree, etc. In theory, multiple sequence alignments can be extended from pairwise sequence alignment, where each pair of sequences gets a score through pairwise alignment, and maximizing the sum of all the pairwise alignment scores will derive the optimal multiple sequence alignments. However, optimizing the multiple alignment score is a NP-complete problem [10] and even dynamic programming needs a time and space complexity $O(L^N)$ for N sequences with length L. Currently many ongoing multiple sequence alignment practices involve tens to hundreds of sequences, which is unimaginable for exhaustive sequence pairing in a reasonable time scale. To overcome the huge computational requirements, heuristic strategies like progressive alignment and iterative alignment are widely adopted in practice, e.g., CLUSTALW [9], T-Coffee [2], etc.

MUSCLE [6] is a new multiple sequence alignment algorithm, it creates alignments with higher accuracy, as well as much faster speed over its predecessors. To make the algorithm more efficient, the latest version of MUSCLE incorporates another sequence aligner - PROBCONS [4], a co-winner of ISMB2004 Best Paper together with MUSCLE, to take advantage of its high alignment accuracy. Though applying a lot of heuristic alignment techniques, MUSCLE suffers from the huge computational intensity, for instance, it takes almost several hours to align 300 sequences with average length of 300 on a commodity PC. The availability of large data sets, typically consisting of thousands of sequences, poses even more challenges in both of the space and execution time. Therefore, parallelization of MUSCLE is particularly important and critical to meet the requirement.

In this paper, we parallelize MUSCLE and characterize its performance on shared memory multiple processor system. The algorithm is firstly broken

Supported by Intel Corporation, the National Natural Science Foundation of China under Grant No. 60273007 and the ChinaGrid.

1-4244-0054-6/06/$20.00

into several independent modules, to clearly express the detailed breakdown and discover the parallel opportunities among these modules. After identifying the concurrency in the whole application as well as in each module, we use OpenMP, a widely used shared memory parallel language, to parallelize the application. The parallel version of MUSCLE makes the algorithm more powerful for handling larger data sets with thousands of sequences, which covers most size range of current multiple sequence alignments. Particularly, it is the first time for parallelized multiple sequence aligner, MUSCLE, to make large protein data set alignment realizable in a short time scale.

The rest of this paper is organized as follows. Section 2 gives an overview of the MUSCLE algorithm. Section 3 presents the application's optimization and parallelization schemes. Section 4 studies the scalability performance of parallel MUSCLE and workload characterizations on shared memory system. Conclusions of the work are presented in Section 5.

2. Algorithm Description

The basic idea of MUSCLE follows the traditional progressive alignment method, to progressively merge two multiple sequence alignments into one. The whole procedure can be described in three steps: (1) calculate the distances between each pair of sequences; (2) use neighbor-joining method [7] and the calculated distances to construct a guide bifurcating tree of which the leaves represent the real sequences; (3) combine the alignments for each pair of leaves and store the result to their father node in the guide tree. This is a repetitive process, where the internal nodes' alignments are merged iteratively into their father node until the root node's alignment is completed.

In practice, MUSCLE uses two-round implementation of progressive alignment. The first round is a basic alignment, the result of which will be used in the second round alignment for refinement. It has three steps, the first one uses k-mer distance to determine the fractional identity between two sequences. Here k-mer stands for a contiguous subsequence of length k. K-mer distance computation takes a relatively low computation, with a time complexity $O(L)$. In the second step, a binary guide tree is constructed through clustering the given distance matrix, where the UPGMA method [8] is applied to assign the distances to a new cluster in building the guide tree. In the last step, the alignment on each node is scored by a log-expectation function [6] and the final alignment maximizes the score with this function. After the three steps, the alignment results will be used as the input in the second round computation.

The second round alignment also follows the three steps of the progressive alignment; however, there is a noticeable difference in the distance computing step. Unlike the simple k-mer distance in the first round alignment, the second round alignment uses a more complex and hence more accurate scheme to generate the distance matrix. Since the distance computation mainly comes from the PROBCONS algorithm, we name it PROBCONS distance in accordance with k-mer distance. PROBCONS distance first calculates posterior probability matrices. Given m sequences, $S = \{s(1), \ldots, s(m)\}$. For each pair of sequences $x, y \in S$ and all $i \in \{1, \ldots, |x|\}$, $j \in \{1, \ldots, |y|\}$, compute the matrix P_{xy}, where equation is the probability that x_i and y_j are paired in a^*, the alignment of x and y is closest to the "true" biological alignment. PROBCONS distance uses a pair-HMM model to simulate the alignment process [4], thereby, these posterior probability matrices are calculated with a modification of the Forward and Backward algorithms for computing posterior probabilities in pair-HMMs as described in Durbin et al. [5].

In practice, the alignment obtained in the first round is used to simplify the matrices computation given the condition that the probability matrices contain a lot of redundant information. There are a lot of heuristics to simplify the computation, e.g., the first letter of sequence X cannot align with the last letter of sequence Y, and their joint probability is zero. MUSCLE defines a possible align area for each letter after the first-round alignment. For every position i in matrix P, MUSCLE only calculates P_{ij} where j belongs to the possible align area of i. Figure 1 depicts the selection process of possible align area for every position i. Since the align area is much shorter than the whole sequence length, we can substantially save the matrices computation time.

After defining the alignment search area, a probabilistic consistency transformation is further applied to the probability matrices, which incorporates similarity of x and y to other sequences from S into the x-y pairwise comparison:

$$P_{xy} \leftarrow \frac{1}{|S|} \sum_{Z \in S} P_{xz} P_{zy}$$

To minimize its complexity, PROBCONS distance

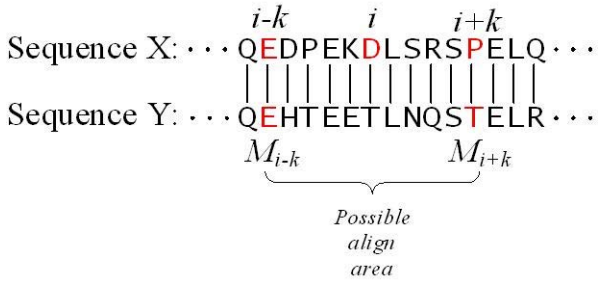

Figure 1. The alignment of sequence X and Y obtained in first round. Assume M_i is the alignment position in sequence Y for position i in sequence X. We define $[M_{i-k}, M_{i+k}]$ as the possible align area for i, where k is a constant and the typical value is 5.

computation step transforms the matrices into sparse matrices by discarding very small value and reduces the time complexity from $O(L^3)$ to $O(cL)$ where c is the average number of non-zero elements per row. Moreover, it also defines the expected accuracy of a pairwise alignment a between x and y to be the expected number of correctly aligned pairs of letters, divided by the length of the shorter sequence:

$$E(a) = \frac{1}{min\{|x|, |y|\}} \sum_{x_i \sim y_j \in a} P(x_i \sim y_j \in a^* | x, y)$$

For each pair of sequences $x, y \in S$, compute the alignment a_{xy} that maximizes expected accuracy by dynamic programming, and define the distance of sequence $x, y = E(a_{xy})$. After the distance refinement, the following steps in the second round progressive alignment are similar to the first round computation, constructing a guide bifurcating tree and combining the alignments for each pair of leaves to update the guide tree.

3. Optimization and Parallelization of MUSCLE

In this section, We describe the strategy of serial optimization and parallelization of MUSCLE. Before that, the MUSCLE program is decomposed into modules and the time distribution of modules is presented.

3.1. Serial MUSCLE Performance

According to previous descriptions in Section 2, MUSCLE is a two round implementation of progressive alignment. The second round includes two primary modules: Probability Matrices Computation (PMC) and Consistency Transformation (CT). The former one calculates posterior probability matrices for every pair of sequences, while the latter one modifies the probability matrices through consistency transformation. In the following sections, we will focus on the first round alignment (FRA) and these two modules.

Figure 2 shows the time distribution of all modules with different datasets. We can easily observe that CT is the most time consuming module. With the increasing size of dataset, the portion of module CT increases dramatically, while that of module FRA remains flat and takes only a small percentage. The algorithmic complexity of PMC and CT are $O(N^2)$ and $O(N^3)$, respectively, where N represents the number of biological sequences. This result matches well with the time breakdown in Figure 2.

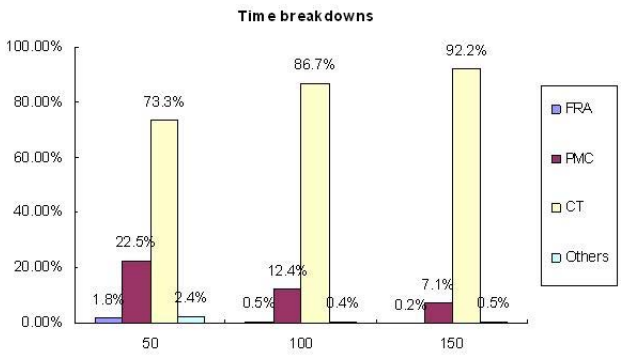

Figure 2. Time breakdown for dataset with size 50, 100 and 150.

3.2. Serial Optimization

In the real implementation, MUSCLE uses STL vector extensively as its primary data structure. Particularly, in module PMC, the distance computation uses one vector for each node pairs, i.e., a large buffer is allocated and initialized within each loop. These frequent *new* and *delete* operations severely degrade the performance and the initialization of the "*new vector*" operation is expensive.

Furthermore, STL containers have their own memory management policies. For vector, when overflow occurs, it will reallocate a new buffer, copy the existing data to it and delete the current one. This operation will cause memory contentions for multi-threaded applications and lead to worse scalability performance.

In our approach, we replace those vectors with some large pre-allocated buffers. Therefore, the frequent *new* and *delete* operations can be eliminated and the memory contentions can also be lessened. At the same time, by reusing those pre-allocated buffers, we can significantly improve the cache locality performance, and benefit a lot from the hardware prefetching technique in Intel's Pentium-4 processor.

Table 1 shows the run time comparison between the PMC module before and after vector replacement. After the optimization, we achieve 1.2x to 2x speedup in the serial and parallel version, respectively. The benefits of reduced memory contentions and better space locality lead to the higher speedup for the parallel version.

module PMC	1 Proc	2 Procs	4 Procs
Original	73.2s	45.0s	37.0s
Optimized	61.1s	34.3s	19.8s

Table 1. Run time of module PMC before and after optimization

3.3. Parallel Implementation

As aforementioned, most of the running time are spent on the three modules (FRA, PMC and CT). Since there are no tight dependencies among these modules, we can only perform parallelization inside each module. Therefore, parallelization of MUSCLE becomes a problem of how to efficiently parallelize these small modules since the underlying algorithm does not allow higher level concurrency.

FRA Module FRA performs the first round alignment. With the guide tree, MUSCLE aligns sequences bottom-up, from leaves to root. This implicates a dependency between a parent node and its children nodes, that each node can be aligned only if its children nodes have already been aligned. This dependency prevents us from fully parallelizing the codes, and gets very little benefit from parallelization. However, the alignments can be considered as a queue of tasks. Once all the alignment tasks of children nodes have been finished, the alignment task of the corresponding parent node is enabled and can be distributed. The tasks of all the enabled nodes don't have any dependencies with each other and can be executed in parallel. In implementation, Intel's *workqueuing* model [12] is used.

PMC and CT Module PMC and CT, which correspond to distance computing steps in the second round alignment, are the most computation intensive parts. Through detailed program profiling, we find that most of the operations are conducted over each pair of sequences, and the pseudo-code is listed below:

$for(i = 0; i < seqnum; ++i)$
  $for(j = i + 1; j < seqnum; ++j)$
    $DoAction()$;

The best way to parallelize these two modules is to assign every processor with the same amount of computations on node pairs. However, the two layer loop prevents us from doing that directly. Since the size of inner loop depends on the outer iteration variable and leads to different amount of works, the simple "*parallel for*" directive will incur severe load imbalance even we use dynamic scheduling policy. Alternatively, we also employ the Intel *workqueuing* model to parallelize this module, where the *taskq* and *task* directive are served to dynamically dispatch the node pair computations to the corresponding processors.

4. Experiment Results and Analysis

The performance measurement of parallel MUSCLE is conducted on a 16-way Intel Xeon shared-memory multiprocessor system. It has 16 x86 processors running at 3.0GHz, 4 levels of cache with each 4MB L4 cache shared amongst 4 CPUs. The sizes of the L1, L2 and L3 caches are 8K, 512K and 4MB respectively. As for the interconnect, the system uses two 4x4 crossbars. We use Intel 8.0 C++ OpenMP compiler tool chain to generate the executables with options -O3 -ipo -openmp, to enable the high levels of compiler optimizations.

The experimental protein sequences data are part of UniProt/Swiss-Prot [15] and are downloaded from EMBL-EBI [13]. Three typical datasets are chosen, which contain 50, 100 and 150 sequences respectively with average length about 330.

To characterize the parallel performance and understand the scalability limiting factors, we investigate the application in different aspects, from the high level general parallel overheads, e.g., synchronizations

penalties, load imbalance, and sequential sections, to the detailed memory hierarchy behavior, including cache miss rates and FSB (Front Side Bus) bandwidth.

4.1. Application Profiling

To study MUSCLE's scalability on systems with multiple processors, we profile some general architecture independent metrics with Intel Vtune thread profiler [14]. As shown in table 2. MUSCLE displays very low imbalance, barrier, locks and synchronization, and is very promising to scale up on system with more processors. However, on system with more than 32 processors, sequential execution time's ratio may adversely impact the scaling performance, according to Amdahl's Law. On the other hand, with the increase of the data set, the influence of the sequential execution time will drop down and ease this problem.

4.2. Performance characterization

In this section, the performance characterization data collected by Intel Vtune Performance Analyzer [14] are presented. Since different modules have different characteristics, we will examine the performance of these modules separately. Dataset 100 is chosen for the characterization.

Cache Miss

Figure 3 shows the local cache miss rates for the whole cache hierarchy on the 16-way system. Compared to module CT, module PMC has higher L2 and L3 cache miss rates. The higher cache miss rate mainly comes from the incontinuous data access pattern in PMC. Typically, the penalty of L3 cache miss costs several hundreds of cycles to fetch the data from the main memory. The last level cache misses directly affect the overall performance, not only in the serial performance but also shows a negative scalability penalty on multi-processor system since all the processors may only share a single bus in the commodity SMP system.

To further study the influence of high L3 cache miss rate, we analyze the program's performance on another 4-way Intel Xeon system with a smaller 2MB L3 cache and no combined L4 cache. Figure 4 shows the cache miss rates on the 4-way system. The L1 and L2 cache misses are almost the same for these two systems. The noticeable difference comes from the L3 cache

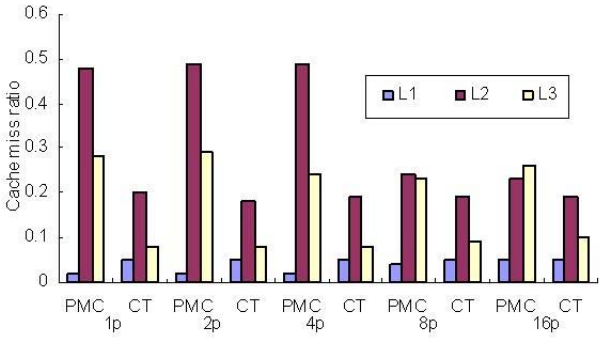

Figure 3. Cache miss rate on 16-way system.

miss rate, where L3 cache misses nearly doubles on the 4-way system. The extreme high L3 cache misses incur tremendous memory traffics and conflicts on the shared bus, and eventually results in high memory access latencies and poor scalability performance for module PMC on the 4-way system. However, on the 16-way system, there are totally 4 clusters of processors, and 4 processors share a large 32M combined L4 cache. The dedicated architecture provides 4x bandwidth and effectively reduces the cost of L3 cache miss.

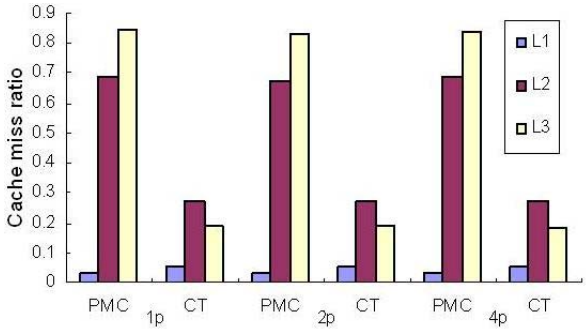

Figure 4. Cache miss rate on 4-way system.

Bandwidth Usage and Memory Latency

Figure 5 shows the bus bandwidth utilization rate with different processor number on the two SMP systems. Generally speaking, memory bandwidth is a key factor which may limit the speedup on multi processors, especially for the shared-bus SMP system. On the 16-way system, the memory bandwidth goes up steadily to 4 processors, and keeps almost constant from 4 to 16 processors. As a result, the bandwidth requirement is far from saturation ($3.2GB/s \times 4$) even with 16 processors. Similarly, in Figure 6, the memory latency on 16-way system remains flat with different processors. In another aspect, it also confirms that

Procs. Number	Parallel	Sequential	Imbalance	Barrier	Lock	Sync
2	98.8%	1.2%	0.003%	0	0	0
4	98.4%	1.6%	0.02%	0	0	0
8	96.8%	3.1%	0.1%	0	0	0
16	94.6%	5.1%	0.3%	0	0	0

Table 2. Statistics of thread profiles

the bandwidth is not saturated on the 16-way system.

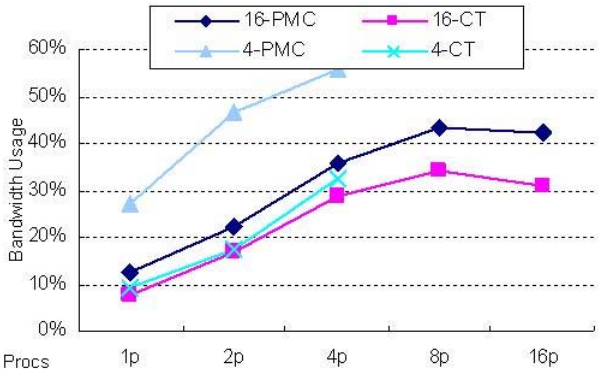

Figure 5. Memory bandwidth usage

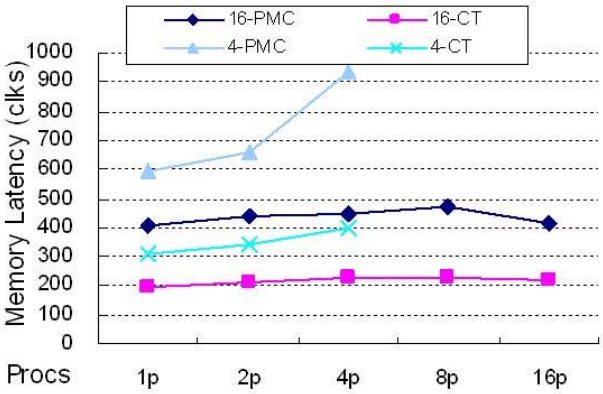

Figure 6. Memory latency

For module PMC on 4-way system, the memory bandwidth curve diverges on 4 processors. The 56% bandwidth utilization rate almost reaches the peak bandwidth which the Pentium-4 system could provide. The extensive memory accesses and bus contentions in turn increase the memory load latency, which is another important indicator of memory wall effect. In Figure 6, module PMC's memory load latency increases sharply from 600 cycles on single-processor to 900 cycles on 4-way system, which directly leads to the poor scalability performance of module PMC on 4-way system.

4.3. Scalability Performance

Figure 7 depicts the overall speedups on 16-way system. We measure the speedup for all three datasets as well as three modules of dataset 100. For the smaller data set, e.g., dataset 50, the speedup curve goes up linearly on 2, 4, and 8 processors, but starts to deteriorate when all the 16 processors are used. The showdown on more processors indicates that the granularity of the work assigned to each processor decreases. With the increase of data set, we get much better speedup curves.

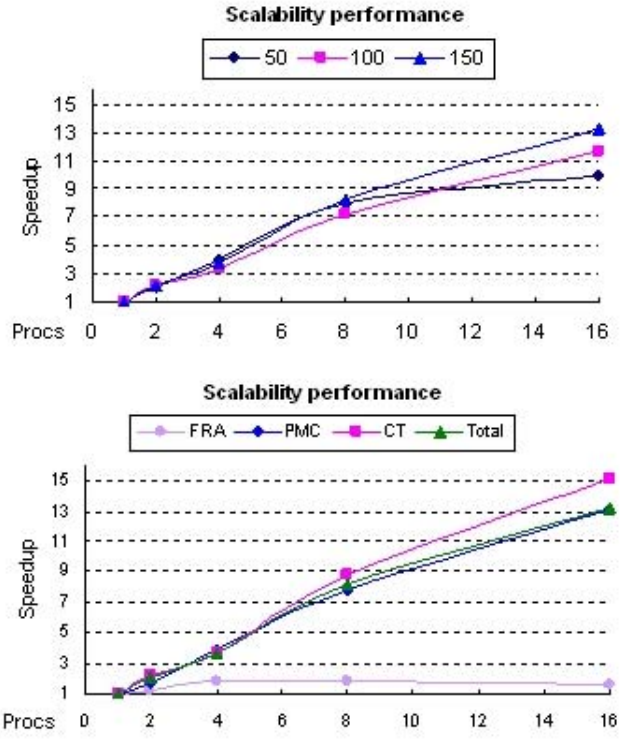

Figure 7. Scalability performance on 16-way system

Sprinkling into the modules, we find FRA, the smallest module, does not scales at all when more than 4 processors are used, the reason of which can be attributed to the strong data dependency among the tree structure and the overhead of *OMP task queue* directives. Nevertheless, it has very little impact on the whole scaling performance on the 16-way system due to its relatively small time percentage. Both Module PMC and CT have abundant parallelism in distributing the tasks among all the processors, and thereby, have fairly good speedup on Unisys 16-way system.

5. Conclusion

MUSCLE is one of the best aligner for multiple sequence alignment, keeping both high accuracy and fast speed. In this paper, we described our implementation of the first parallel version of MUSCLE and studied its performance on shared memory system. The experimental results show that our parallel implementation scales pretty well on 16-way system (15.2x speedup on dataset 100). By comparing with the performance of a 4-way system, we conclude that large L3 cache as well as combined L4 cache can sharply reduce the last level cache miss rate, which is the bottleneck of MUSCLE, and thus do help to improve the scalability performance of MUSCLE.

References

[1] S. F. Altschul and D. J. Lipman. Trees, stars, and multiple sequence alignment. *SIAM Journal on Applied Mathematics*, 49(1):197–209, Feb. 1989.

[2] D. H. C Notredame and J. Heringa. T-coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 320:205–217, 2000.

[3] H. Carillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, Oct 1988.

[4] C. B. Do, M. S. Mahabhashyam, M. Brudno, and S. Batzoglou. PROBCONS: Probabilistic consistency-based multiple sequence alignment. *Genome Research*, 15:330–340, Feb. 2005.

[5] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis. Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[6] R. C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, Mar. 2004.

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

[8] P. H. A. Sneath and R. R. Sokal. *Numerical Taxonomy : The principles and practice of numerical classification*. W. H. Freeman, San Francisco, 1973.

[9] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, 22:4673–4680, 1994.

[10] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[11] M. S. Waterman. *Introduction to computational biology: Maps, sequences and genomes*. Chapman & Hall, 1995.

[12] Compiler support for the workqueuing model extends OpenMP. http://www.intel.com/cd/ids/developer/asmona/eng/newsletter/57769.htm?page=1.

[13] European bioinformatics institute. http://www.ebi.ac.uk/.

[14] Intel corp.: Intel vtune performance analyzer. http://developer.intel.com/software/products/vtune/.

[15] The universal protein resource. http://www.ebi.uniprot.org/index.shtml.