1

Parallel Implementations of Local Sequence Alignment: Hardware and Software

Vipin Chaudhary, Feng Liu, Vijay Matta,
Xiandong Meng, Anil Nambiar,
Ganesh Yadav
Parallel & Distributed Computing Laboratory,
Wayne State University, Detroit, MI, USA.

Laurence T. Yang
Department of Computer Science
St. Francis Xavier University,
Antigonish, NS, B2G 2W5, Canada.

1.1 INTRODUCTION

With the progress in computer science and raw computational power available today, human quest to learn and understand the complex relationships within the subsets of biology, *e.g.*, biological response, biodiversity, genetics, medicine, etc. has got a new promise. Computational Biology represents the marriage of computer science and biology, and spans may disciplines, such as bioinformatics (genomics and post-genomics), clinical informatics, medical imaging, bioengineering, etc. It finds application in many areas of life science, *e.g.*, the development of human therapeutics, diagnostics, pyrognostics and forensics, up through the simulation of large entities such as populations and ecosystems.

Genomics is the determination of the entire DNA sequence of an organism. The goal of modern human genomics is preventive, predictive, and individualized medicine. In agriculture, the goal is the production of foods with improved production characteristics and, increasingly beneficial consumer traits. Post-genomics refers to the biological processes that follow from DNA sequence(e.g. transciptomics, proteomics, metabolomics, etc.).

Modern biopharm companies currently have about $2 - 10$ Terabytes of genomics data, large phrama have $10 - 40$ Terabytes, and genomic data providers have over 100 Terabytes of data. The genomics effort is to collect and process terabytes of such heterogeneous and geographically disperse data into *information* (translating the spots into the sequence of nucleotide bases A,C,G,T comprising sequences of DNA, or the 3-D structure of a protein with the relative location of all of its atoms), then *knowledge* (the location of genes in the sequence, the proteins that come from the genes, the function of the genes and proteins, how they interact, their relationship to each other, etc.), and then take *action* (the simple decision to run another experiment, or the decision to risk $2 Billion to develop the drug candidate). Eventually they ship the information to the regulatory agencies. Only after approval does the company become a manufacturer of a tangible product (drug, therapy, diagnostic, etc.).

While CPU architectures are struggling to show increased performance, the volume of biological data is greatly accelerating. For example, Genbank, a public database of DNA, RNA and protein sequence information, is doubling about every 6 months. In keeping up with the Moore's law the density of the transistors on a chip has doubled for four decades, but may slow, according to Semiconductor Industry Association, as certain physical limits are reached. Also DRAM speeds have not kept up with the CPU speeds thus hitting the *"memory wall"*. Most science and engineering problems tend to plateau beyond 8 CPUs. However, genomics algorithms can achieve much better parallelization (sometimes called "embarrassingly parallel") because they can be deconstructed into a large number of independent searches with little message passing, or coordination, between jobs/threads if the data is appropriately passed to the appropriate processors. The final results are then assembled when the independent jobs are completed.

1.1.1 Types of Sequence Alignment

Sequence alignment refers to the procedure of comparing two or more sequences by searching for a series of characters (nucleotides for DNA sequences or amino acids for protein sequences) that appear in the same order in the input sequences. Although residues are mostly used to refer to amino acids, for brevity purposes, residues will be used to imply both nucleotides and amino acids in the remainder of this discussion. A distinction will be made when necessary. The sequence alignment problem is often referred to as the longest common substring problem. Regions in the new sequence and the known sequence that are similar can help decipher biological functions or evolutionary information about the new sequence. The alignment of two or more sequences is anchored around the longest common substring and the remaining, non-matching residues can represent gaps, insertion or deletion. When aligning multiple sequences, the goal is to discover signatures or motifs that are common to all the sequences. A motif or a signature is a sequence of residues that is common to the aligned sequences and can help identify a family of nucleic acid sequences or protein sequences.

The alignment of two sequences (pairwise alignment), or multiple sequences (multiple alignment), and the alignment of short or long sequences such as an entire genome may require different types of algorithms. The algorithms used in all of these four cases can be either dynamic programming based or heuristic-based or a combination of both. Dynamic programming is a general optimization technique that relies on the fact that the solution to a problem consists of the combined solutions of the sub-problems. Furthermore, several of the sub-problems may be the same. Thus, they are solved only once. Dynamic programming based algorithms generate optimal solutions. However, they are computationally intensive which makes them impractical for a large number of sequence alignments. A more practical solution is one that uses a heuristic to generate a near optimal solution. Heuristics are approximation algorithms. In the case of sequence alignment, these heuristics often use a combination of a restricted form of dynamic programming (e.g., dynamic programming is only used for a small subset of the residues in a sequence rather than on the entire sequence) and other approximations in order to reduce the search space of possible solutions.

1.1.2 Pairwise Alignment

Pairwise alignment is the alignment of two sequences. In general, the purpose of this alignment is to extract the sequences that are similar (homologous) to a given input sequence from a database of target sequences. That is, the input sequence is aligned with each target sequence in the database and the top ranking sequences represent the sequences with the highest level of similarity to the input sequence. The input sequence is also called the query sequence. Each alignment between the input sequence and a target sequence is one pairwise alignment. A score is associated with each pairwise alignment in order to indicate the level of similarity between the query sequence and the corresponding target sequence. This score is determined based on a scoring matrix and specific penalties for insertions, deletions and gaps. The scoring matrix represents the weight associated with a match for all different types of nucleotides or amino acids.

1.1.3 Multiple Sequence Alignment

In multiple sequence alignment, the objective is to find a common alignment for multiple sequences. Several approaches for multiple sequence alignment have been proposed. Initial implementations were based on an extension of the Smith Waterman algorithm to multiple sequences. This implementation, which is based on dynamic programming, generates an optimal solution, but is computationally very intensive. More recent approaches incrementally build multiple sequence alignment by using heuristics.

1.2 SEQUENCE ALIGNMENT PRIMER

1.2.1 Parallel Programming Models

Parallel algorithms for analyzing DNA and protein sequences are becoming increasingly important as sequence data continues to grow. In simple terms, parallel software enables a massive computational task to be divided into several separate processes that execute concurrently through different processors to solve a common task [osw01].

In particular, two key features can be used to compare models: granularity and communication. Granularity is the relative size of the units of computation that execute in parallel, *e.g.* fineness or coarseness of task division; Communication is the way that separate units of computation exchange data and synchronize their activity.

1.2.1.1 Coarse & Fine-Grain parallelism The finest level of software granularity is intended to run individual statements over different subsets of a whole data structure. This concept is called data-parallel, and is mainly achieved through the use of compiler directives that generate library calls to create lightweight processes called threads, and distribute loop iterations among them. A second level of granularity can be formulated as a "block of instructions". At this level, the programmer identifies sections of the program that can safely be executed in parallel and inserts the directives that begin to separate tasks. When the parallel program starts, the run-time support creates a pool of threads, which are unblocked by the run-time library as soon as the parallel section is reached. At the end of the parallel section, all extra processes are suspended and the original process continues to execute.

Ideally, if we have n processors, the run time should also be n times faster with respect to the wall clock time. In reality, however, the speedup of a parallel program is decreased by synchronization between processes, interaction, and load imbalance. In other words, the overhead to coordinate multiple processes require some time added to the pure computational workload.

Much of the effort that goes into parallel programming involves increasing efficiency. The first attempt to reduce parallelization penalties is to minimize the communication cost between parallel processes. The simplest way, when possible, is to reduce the number of task divisions; in other words, to create coarsely-grained applications. Once the granularity has been decided, communications are needed to enforce correct behavior and create an accurate outcome.

1.2.1.2 Inter Process Communication When shared memory is available, inter-process communication is usually performed through shared variables. When several processes are working over the same logical address space, locks, semaphores or critical sections are required for safe access to shared variables.

When the processors use distributed memory, all inter-process communication must be performed by sending messages over the network. The message-passing paradigms, *e.g.*, MPI, PVM, etc., are used to specify the communication between a set of processes forming a concurrent program. The message-passing paradigm is attractive because of its wide portability and scalability. It is easily compatible with both distributed-memory multi-computers and shared-memory multiprocessors, clusters, and combinations of these elements. Writing parallel code for a distributed memory machines is a difficult task, especially for applications with irregular data-access patterns. To facilitate this programming task, software distributed shared memory provides the illusion of shared memory on top of the underlying message-passing system [roy89].

Task scheduling strategies suggest that obtaining an efficient parallel implementation is fundamental to achieving a good distribution for both data and computations. In general, any parallel strategy represents a trade-off between reducing communication time and improving the computational load balance.

A simple task scheduling strategy is based on a Master-Slave approach. The Master-Slave paradigm consists of two entities: a master and multiple workers. For coarse grain parallelism, the database is divided into blocks of sequences. These blocks can be assigned to the slaves following the work pool approach with dynamic load balancing. When one slave finishes generating all the pairwise alignments for the target sequences in its block, another block is assigned to it. This process is continued until all the sequences in the database are processed. The number of blocks is usually orders of magnitude higher than the number of processors, and blocks are assigned to processors dynamically. This dynamic load balancing approach is more efficient than a static load balancing approach since the execution time associated with the pairwise alignment is not known *a priori* and can vary from a pair of sequences to the next. The factors that have an impact on the execution time required by a pairwise alignment include the length of the two sequences and how similar they are. The results generated from the individual slaves have to be combined and sorted according to the score calculated for each pairwise alignment. In order to perform this task, the slaves can send their results to the master, which take care of generating this final result. Usually, the communication takes place only between the master and the workers at the beginning and at the end of the processing of each task.

1.2.2 Parallel Computer Architectures

The algorithms for database searching can be implemented to run efficiently on various types of hardware with the ability to perform several operations simultaneously. There is a wide range of different hardware available on which the algorithms can be implemented. Hughey [hug96] has reviewed various types of hardware that can be used and their performance. The hardware can be divided into a group of general purpose computers, which can be used for many different kinds of computations, and a group of hardware specifically designed for performing sequence alignments and database searches.

1.2.2.1 General-purpose Parallel Computers

General purpose computers with parallel processing capabilities usually contain a number of connected processors, ranging from dual-CPU workstations to supercomputers. The well-known dynamic programming or heuristic algorithms must be rewritten to run on such computers. The algorithms can be parallelized at different scales, from a simple coarse-grained parallelization where *e.g.*, the database sequences are divided on two or more processors, each comparing the sub-database sequence to the query sequence, to a complicated fine-grained parallelization where the comparison of the query sequence against one database sequence is parallelized. The speed gained varies according to the type of algorithm and computer architecture.

A cluster of workstations (either single- or multi-CPU) connected by an Ethernet network is loosely connected processors, is very interesting for sequence database searches, because of the independence between the different sequences in the database.

Microparallelism can be classified into SIMD (Single-Instruction, Multiple-Data) and MIMD (Multiple-Instruction, Multiple-Data) types according to whether the processing units perform the same or different operations on their data. It is an interesting form of SIMD, where a 128-bit wide integer register of a CPU is divided into sixteen smaller 8-bit units, and where the same arithmetic or logical operation can be performed simultaneously and independently on the data in each of the individual units. This technique can be performed on ordinary CPUs using normal instructions combined with a technique involving masking of the high order bits in each unit. However, it has become much easier recently with the introductions of MMX/SSE from Intel, MMX/3DNow from AMD and VIS from SUN, which allows fine grain parallelism to be exploited for a single pairwise alignment.

1.2.2.2 Special-purpose Parallel Hardware A number of different designs for special-purpose hardware for performing sequence alignments and database searching have been proposed and implemented. Their advantage over general-purpose computers is that they can be tailored specifically to perform sequence comparisons at a high speed, while the disadvantage is high cost.

Special-purpose hardware is usually built using either FPGA (Field-Programmable Gate Arrays) or custom VLSI (Very Large Scale Integration) technology. The advantage of FPGA is that they are reprogrammable and can be built to work in a given function, and hence can be changed to remove bugs or to work with different algorithms, while VLSI is customarily designed to a very specific purpose and cannot be changed. The advantage of VLSI is a lower cost per unit (at least in large volumes) and a higher processing speed. However, the design and initial costs for VLSI systems are higher than for FPGA.

1.2.3 Local Sequence Alignment Sotfware

1.2.3.1 Sequence Alignment Parallelization Sequence alignment is the most widely used bioinformatic application. It is also one of the most familiar applications to begin a discussion about parallelization in bioinformatics. Sequence alignment has a very simple form as far as data flow is concerned, and a broad range of strategies have been proposed to apply parallel computing.

Searching on DNA or protein databases using sequence comparison algorithm has become one of the most powerful technique to help determine the biological function of a gene or the protein it encodes. The primary influx of information for database searching is in the form of raw DNA and protein sequences. Therefore, one of the first steps towards obtaining information from a new biological sequence is to compare it with the set of known sequences contained in the sequence databases, using algorithms such as BLAST [alt97], Needleman-Wunsch [wun70] and Smith-Waterman [smw81] algorithm. Results often suggest functional, structural, or evolutionary analogies between the sequences.

Two main sets of algorithms are used for pairwise comparison: exhaustive algorithms and heuristic-based algorithms. The exhaustive algorithms based are on dynamic programming methodology such as Needleman-Wunsch [wun70] and Smith-Waterman [smw81] algorithm. The heuristic approaches are widely used such as the FASTA [pea90]and BLAST [alt97] families. Most of the currently used pairwise alignment algorithms are heuristic based.

The first widely used program for database similarity searching was FASTA [pea90]. FASTA stands for FAST-All, reflecting the fact that it can be used for a fast protein comparison or a fast nucleotide comparison. This program achieves a high level of sensitivity for similarity searching at high speed. The high speed of this program is achieved by using the observed pattern of word hits to identify potential matches before attempting the more time consuming optimized search. The trade-off between speed and sensitivity is controlled by the k-tuple parameter, which specifies the size of the word. Increasing the k-tuple decreases the number of background hits. The FASTA program does not investigate every word hit encountered, but instead looks initially for segments containing several nearby hits. By using a heuristic method, these segments are assigned scores and the score of the best segment found appears in the output. For those alignments finally reported, a full Smith-Waterman alignment search is performed.

BLAST [alt97] is another heuristic-based algorithm for sequence homology search. As in FASTA, it finds database sequences that have k consecutive matches to the query sequence. The value of k is 3 for protein sequence and 11 for DNA sequence. Several variations [zha00, alt97] of the original BLAST algorithm were developed to accommodate different types of sequence alignments. For example, MEGABLAST uses the XDrop alignment algorithm [zha00]. It is particularly tuned for the alignment of two DNA sequences that are highly similar. This algorithm is computationally efficient because it considers long runs of identical adjacent nucleotides. If the two sequences differ by 3%, the expected length of the run is 30 nucleotides. The algorithm is also computationally efficient because it completely avoids the use of dynamic programming even in a limited context. It uses a greedy algorithm instead. A greedy algorithm is one type of a heuristic that is developed with the assumption that a global optimal can be obtained by making a sequence of local optimal decisions, whereas dynamic programming is a global optimization algorithm. XDrop was used to align entire genomes and it was found [zha00] to be 10 times faster than BLAST for long and highly similar sequences.

PSI-BLAST [alt97] executes several iterations of the BLAST algorithm. However, the scoring matrix, which is used to score the pairwise alignment, is not fixed in PSI-BLAST. The scoring matrix includes the weights corresponding to a match for all types of nucleotides or amino acids. After every iteration the top ranking target sequences in the pairwise alignment are used to update the scoring matrix. Also, PSI-BLAST uses a position-specific scoring matrix where two matching residues are assigned a score based not only on the importance of the match but also on the position of the residue in the sequence. PSI-BLAST is more sensitive than BLAST in detecting weak sequence similarities.

Regardless of the algorithm used, in the case of pairwise alignment, an input sequence is aligned against a list of target sequences from a database resulting in multiple pairwise alignments. In each pairwise alignment, one of the two sequences is the input sequence and the other is one sequence from the database. This process can be parallelized in two ways: 1) multiple pairwise alignments can be executed in parallel (coarse grain parallelism) and 2) a parallel version of the alignment algorithm can be used to speed up each individual pairwise alignment (fine grain parallelism).

Given a set of input sequences, ClustalW [che03] implements multiple alignments using a tree-based method. Pairwise alignments are first constructed for each pair of sequences from the input set. These alignments are used to construct a similarity matrix. Each entry in the matrix represents the similarity distance between any two sequences from the input set. The similarity matrix is used to construct a tree that will guide the multiple sequence alignment. Closely related sequence pairs are aligned first resulting in partial alignments. These partial alignments are then either combined with other neighboring partial alignments or sequences in the guiding tree. The computational complexity of Clustal W is reduced from being exponential when dynamic programming based multiple alignment is used to a second order polynomial.

For n sequences the number of comparisons to be made are $n(n-1)/2$ which is very large as the number of sequences increases. This pairwise comparison can be done in parallel. There are different approaches such as ClustalW-MPI and SGI's HT ClustalW and MULTICLUSTAL. These approaches increase the speed of aligning multiple sequences. ClustalW-MPI and HTClustalW will be discussed in detail in the following sections.

The demand for computational power in the bioinformatic field will continue to grow as the complexity and the volume of data increases. This computational power can only be delivered by large-scale parallel computers that either have distributed memory architecture or shared memory architecture. Distributed computing has been already used successfully in sequence alignment. In general, most of these applications have been implemented by using the work pool approach with coarse grain parallelism. This type of implementation is ideal for clusters built with off-the-shelve personal computers. Sequence Alignment is expected to continue to draw increasing attention and it will drive several high performance computing efforts.

1.3 SMITH-WATERMAN ALGORITHM

When looking for similarities between subsequences of two sequences, as is usually the goal in the methods used to find homologies by database searches, a local alignment method is more appropriate than a global. The simple dynamic programming algorithm described by Smith and Waterman [smw81] is the basis for this type of alignments. The Smith-Waterman algorithm is perhaps the most widely used local similarity algorithm for biological sequence database searching.

In Smith-Waterman database searches, the dynamic programming method is used to compare every database sequence to the query sequence and assign a score to each result. The dynamic programming method checks every possible alignment between two given sequences. This algorithm can be used both to compute the optimal alignment score and for creating the actual alignment. It uses memory space proportional to the product of the lengths of the two sequences, mn, and computing time proportional to $mn(m+n)$. The recursion relations used in the original Smith-Waterman algorithm are the following:

$$H_{i,j} = max\{H_{i-1,j-1}, S[ai, bj], E_{i,j}, F_{i,j}\}$$
where

$$E_{i,j} = max_{0<k<i}\{H_{i-k,j} - g(k)\}$$

$$F_{i,j} = max_{0<l<j}\{H_{i,j-l} - g(l)\}$$

Here, $H_{i,j}$ is the score of the optimal alignment ending at position (i, j) in the matrix, while $E_{i,j}$ and $F_{i,j}$ are the scores of optimal alignments that ends at the same position but with a gap in sequence A or B, respectively. S is the match/mismatch value of ai and bj, or amino acid substitution score matrix, while g(k) is the gap penalty function. The computations should be started with $E_{i,j}$ = $F_{i,j}$ = $H_{i,j}$ = 0 for all $i = 0$ or $j = 0$, and proceeded with i going from 1 to m and j going from 1 to n (see Figure 1.1).

The order of computation is strict, because the value of H in any cell in the alignment matrix cannot be computed before all cells to the left or above it has been computed. The overall optimal alignment score is equal to the maximum value of $H_{i,j}$.

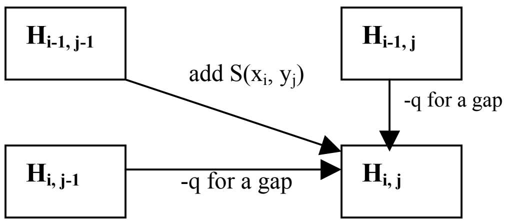

Fig. 1.1 Dynamic programming illustration

Gotoh [got82] reduced the time needed by the algorithm to be proportional to mn when affined gap penalties of the form $g(k) = q + rk$;($q \geq 0$, $r \geq 0$)are used, where q is the gap opening penalty and r is the gap extension penalty. When only the actual

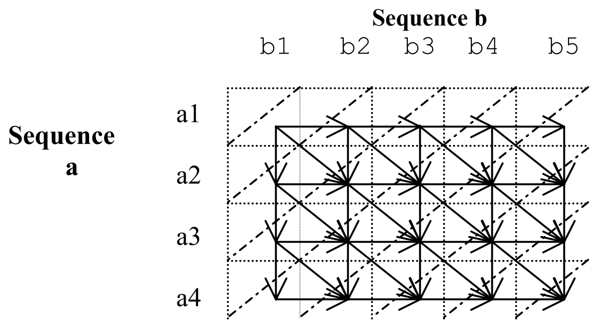

Fig. 1.2 Data dependency in Smith-Waterman alignment matrix

optimal local alignment score is required, the space requirements were reduced to be proportional to the smallest of m and n. The new recursion relations for $E_{i,j}$, and $F_{i,j}$ are as follows:

$$E_{i,j} = max_{0<k<i}\{H_{i-1,j} - q + E_{i-1,j} - r\}$$

$$F_{i,j} = max_{0<l<j}\{H_{i,j-l} - q + E_{i,j-1} - r\}$$

All the searching process can be divided into two phases [men04]. In the fist phase, all the elements of two sequences have to be compared and form a scoring matrix. Following the recurrence equation in Figure 1.1, the matrix is filled from top left to bottom right with each entry $H_{i,j}$ requiring the entries $H_{i-1,j}$, $H_{i,j-1}$, and $H_{i-1,j-1}$ with gap penalty $q = r$ at each step. Once scores in all cells are calculated, the second phase of the algorithm identifies the best local alignments. Since they might be biologically relevant, alignments with score value above a given threshold are reported. Thus, for each element of the matrix a backtracking procedure is applied to find out the best local alignment.

Figure 1.2 shows the data dependencies in Smith-Waterman algorithm. As mentioned in the previous section, there are three possible alignments to choose from when calculating one element: alignment of the symbol in the row considered with gap – horizontal arrow, alignment between the symbols in the row and column considered with match or mismatch - diagonal arrow, and alignment of the symbol in the column considered with a gap - vertical arrow. This means that rows or columns can't be computed in parallel. The only elements on each successive anti-diagonal (labelled dashed line in Figure 1.2 are processed in parallel. These data dependencies present a serious challenge for sufficient parallel execution on a general-purpose parallel processor.

1.3.1 Parallel Computation Approach for Smith-Waterman Algorithm

Database searching applications allow two different granularity alternatives to be considered: fine–grained and coarse-grained parallelism. Early approaches focused on data-parallel over SIMD machines.

1.3.1.1 Fine-grain parallelism Typical dynamic programming-based algorithms, like Smith-Waterman algorithm, compute an $H_{m,n}$ matrix (m and m being the sequence lengths) depending on the three entries $H_{i-1,j}$, $H_{i,j-1}$, and $H_{i-1,j-1}$. Fine-grain means that processors will work together in computing the H matrix, cell by cell. Some researchers organized the parallel machine as an array of processors to compute in diagonal-sweep fashion the matrix H (see Figure 1.3). An advantage is that this strategy only requires local communications in each step. PE_i sends $H_{i,j}$ to PE_{i+1} to allow it to compute $H_{i+1,j}$ in the next step, while PE_i computes $H_{i,j+1}$. Query sequence length determines the maximum number of processors able to be assigned, and processors remain idle at begin/end steps.

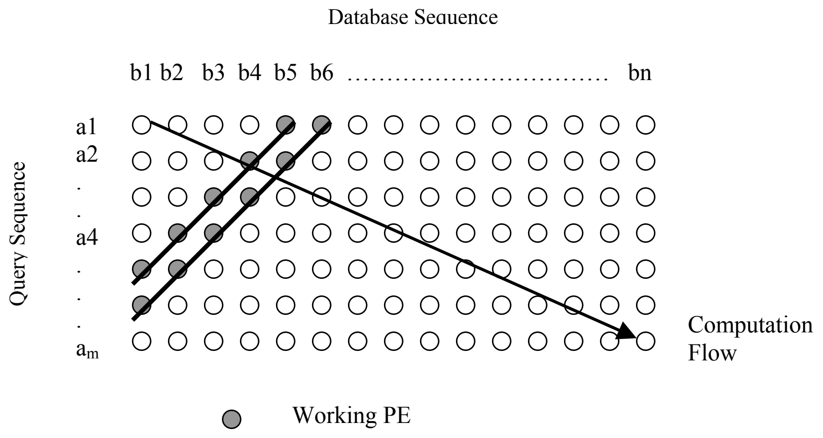

Fig. 1.3 Diagonal-sweep fine-grained workload distribution for multiprocessors machines to avoid data dependencies in Smith-Waterman algorithm

Rognes [rog00] implemented the Smith-Waterman algorithm using Intel's MMX/SSE technology. Six-fold speed-up relative to the fastest known Smith-Waterman implemantation on the same hardware was achievied by optimized 8-way parallel processing approach.

1.3.1.2 Coarse-grain Parallelism There are several proposed strategies for achieving coarse-grained parallelism in sequence alignment applications. Most of them can be explained on the basis of the flow chart shown in Figure 1.4. First of all, the program sets the initial stage of the algorithm. Next, it manages the algorithm extension, which works until the number of database sequences is exhausted, then, fetches the next database sequence to be compared against the query sequence. The result value is saved to rank the best results as in the following step. Finally, statistical significance can incorporate a optimization process and the last step is to output results.

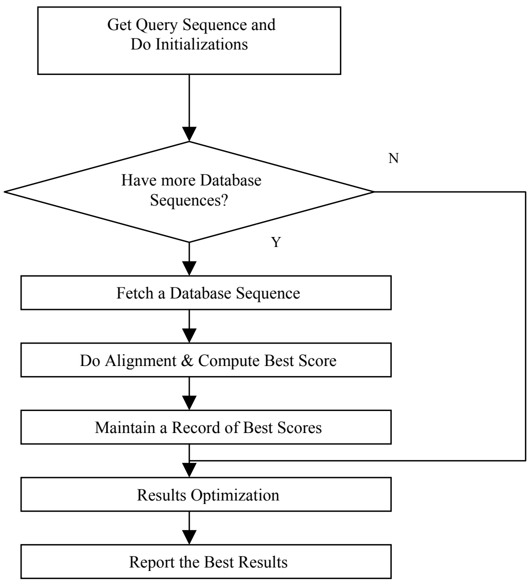
Get Query Sequence and Do Initializations
Have more Database Sequences?
N
Y
Fetch a Database Sequence
Do Alignment & Compute Best Score
Maintain a Record of Best Scores
Results Optimization
Report the Best Results

Fig. 1.4 Sequential flow chart for a sequence database searching application

As should be noted, the algorithm has a very simple form as far as data flow is concerned. The database sequence corresponds to the data set to be searched, which is a set of sequences of different lengths. In essence, in a typical coarse-grained parallel implementation, one of the processors acts as a "master", dispatching blocks of sequences to the "slaves" which, in turn, perform the algorithm calculations. When the slaves report results for one block, the master sends a new block. This strategy is possible because results from the comparison between query and database sequences are independent of the previous results deriving from the comparison of the query with other sequences.

However, the time required in the processing of any given sequence depends not only on the length of the sequence, but also on its composition. Therefore, the use of a dynamic load balancing strategy is necessary. The simplest way is to modify the way in which the master processor distributes the load on demand from the slaves. Obviously, sending one-sequence messages introduces additional expensive time overhead due to the high number of messages interchanged. Thus, rather than distributing messages sequence-by-sequence, better results are achieved by dispatching blocks of sequences.

1.4 FASTA

FASTA [pea90] finds homologous sequences by using a four-step process. First, sequences that have subsequences of at least k adjacent residues that match subsequences in the query sequence are identified. The recommended value of k for a protein sequence alignment is 2 and for a DNA sequence alignment is between 4 and 6. The second step combines groups of these matching subsequences into longer matching regions called initial regions. Each initial region consists of one or more matching subsequence separated by mismatching regions of residues. Gaps are not allowed within the mismatching regions. That is, the number of residues between two consecutive matching subsequences within the same initial region has to be the same in the query sequence and the target sequence. These initial regions are scored and the best 10 initial regions are selected. During the third step, dynamic programming is used to combine nearby initial regions and new scores are assigned to the combined regions. This is an example of how dynamic programming is used in a limited context (i.e., only for nearby initial regions) within heuristics. In this third step mismatching regions between the initial regions may contain gaps. The scores generated in the third step are used to rank the database sequences. During the fourth step, the Smith-Waterman [smw81] algorithm is applied to the top ranking sequences from the previous step. Specifically, this algorithm is used to align the query sequence and the database sequences within the selected initial regions and their neighboring residues. The fourth step in FASTA is another example of the use of dynamic programming in a limited context within a heuristic based alignment algorithm. The Smith-Waterman [smw81] algorithm is only used for top ranking sequences and only within selected regions of these sequences. Limiting the use of dynamic programming increases the computational efficiency of the alignment algorithm. However, it also means that the generated solution is only a sub-optimal solution rather than an optimal one.

FASTA provides different executables for different types of sequence alignment.

- FASTA: Nucleotide sequence / Nucleotide sequence database.
- SSEARCH: Protein sequence / Protein sequence database.
- TFASTA : protein sequence / Six-frame translations of a nucleotide sequence database (treats each frame separately).
- FASTX: Six-frame translations of a nucleotide sequence / Protein sequence database.
- TFASTX: Protein sequence / Six-frame translations of a nucleotide sequence database (treats the forward or reverse three frames as one sequence).

1.4.1 Parallel Implementation of FASTA

To study the performance of parallel FASTA, a large quantity of research work has been done. The parallel efficiency of FASTA programs on Sun servers [sha01] can be quite high, especially for large searches. The experiment results show a 54-fold speedup of FASTA search when it runs on 62 CPUs of the Sun Enterprise 10000 (64 400 MHz CPUs with 8MB L2 cache). Parallel scaling for smaller queries is typically much lower.

The performance of the FASTA programs was studied on PARAM 10000 [jan03], a parallel cluster of workstations. The FASTA program executes very quickly when a small query or database is chosen, but becomes compute intensive when searching a query of longer length against huge databases. The parallel FASTA ported using Sun-MPI libraries was used to run on Fast Ethernet across 2 to 64 processors and a speedup of 44 fold was observed on 64 processors. While searching a longer query sequence against a huge database using parallel FASTA, better speedup was observed than with smaller query lengths. Thus, parallel FASTA can be more effectively used when long genome sequences of human chromosomes, that is, those having more than 10 mega bases, need to be searched against large genome sequence databases.

1.5 BLAST

BLAST [alt90] or Basic Local Alignment Search Tool is a heuristic based search algorithm to match sequences. This heuristic search method seeks words of length W that score at least T when aligned with the query and scored with a substitution matrix. Words in the database that score T or greater are extended in both directions in an attempt to find a locally optimal un-gapped alignment or HSP (high scoring pair) with a score of at least S or an E value lower than the specified threshold. HSPs that meet these criteria will be reported by BLAST, provided they do not exceed the cutoff value specified for number of descriptions and/or alignments to report.

In the first step, BLAST uses words of length w instead of k-tuples. These words also include conservative substitutions. The words used in BLAST contain all w-tuples that receive a score T, above a certain level, when compared using the amino acid substitution matrix. By default, BLAST uses $w = 3$ and $T = 11$. A given triplet in the query sequence will then match the triplets in the database sequence that has a score of 11 or more when the three pairs of amino acids are compared.

In the second step, BLAST extends the initial words into so-called High-scoring Segment Pairs (HSPs) using the amino acid substitution matrix. This extension is performed in both directions along the diagonal from the initial word and is stopped when the potential score falls a level X below the currently found maximum score of the HSP.

It was found that 90% of the time was spent in extending the word and most of this extension wouldn't lead to a HSP. It was also found that most of the HSPs found would have multiple hits. So rather than extending the word on a single hit, one would only extend a word if there were multiple hits. To keep the probability of finding a similarity constant, one reduces the threshold T. This is known as the Two-Hit alignment method. In addition, the first version of BLAST does not consider gapped alignments at all, but computes a statistical measure of significance based on the highest scoring HSPs using sum-statistics [kar90].

Altschul et. al. [alt97] describe version 2 of NCBI BLAST, which includes a few improvements, that increases both the speed and the sensitivity of the program. In the first step, BLAST 2 uses the Two-Hit alignment method to improve performance. This double-hit method not only reduces the number of hits substantially, but also reduces sensitivity relative to the first version of BLAST. The extension of HSPs in the second step is performed in the same manner as with the previous version although with far fewer HSPs, and hence much faster. Using midpoints on the HSPs as seeds, BLAST 2 performs an accurate gapped alignment constrained not to contain any low-scoring regions. This gapped alignment leads to much increased sensitivity over the original BLAST program. The alignments take a lot of time and are hence only performed for the HSPs scoring 40 or above, which represents only about 2% of the database sequences. NCBI BLAST 2 uses the new statistics for gapped alignments described by [alt96] to compute an E-value expressing the expected number of random matches in the database having a given score.

To utilize these heuritics for sequence alignments, NCBI provides different executables for different types of sequence alignment. Blastp is used for matching protein query sequence against protein database. Blastn is used for matching neucleotide query sequence against neucleotide database. Blastx is used for matching nucleotide query sequence translated in all reading frames against a protein sequence database. Tblastn is used for matching protein query sequence against a nucleotide sequence database dynamically translated in all reading frames. Tblastx is used for matching the six-frame translations of a nucleotide query sequence against the six-frame translations of a nucleotide sequence database. Though they are different programs, they all use the same Two-Hit heuristic for the comparison of query against the database.

1.5.1 TurboBLAST

TurboBLAST TurboBLAST [bjo02] is an accelerated, parallel deployment of NCBI BLAST, which delivers high-performance, not by changing the BLAST algorithm, but by coordinating the use of multiple copies of the unmodified serial NCBI BLAST application on networked clusters of heterogeneous PCs, workstations, or Macintosh computers. As a result, TurboBLAST supports all of the standard variants of the BLAST algorithm supported in NCBI BLAST (blastn, blastp, blastx, tblastn, and tblastx). It provides results that are effectively identical to those obtained with the NCBI application.

An individual BLAST job specifies a number of input query sequences to be searched against one or more sequence databases. In order to achieve parallel speed-up, TurboBLAST implements a distributed Java "harness" that splits BLAST jobs into multiple small pieces, processes the pieces in parallel, and integrates the results into a unified output. The harness coordinates the following activities on multiple machines:

- Creation of BLAST tasks, each of which requires the comparison of a small group of query sequences (typically 10-20 sequences) against a modest-sized partition of one of the databases sized so that the entire task can be completed within the available physical memory without paging.
- Application of the standard NCBI blastall program to complete each task and
- Integration of the task results into a unified output.

This approach has the advantage that it is guaranteed to generate the same pairwise sequence comparisons as the serial version of BLAST since it uses exactly the same executable to perform the search computations. High performance is achieved in two ways. First, the size of each individual BLAST task is set adaptively so that blastall processing will be efficient on the processor that computes the task. Second, a large enough set of tasks is created so that all the processors have useful work to do and so that nearly perfect load balance can be achieved.

Initial benchmarks of TurboBLAST on a network of 11 commodity PCs running Linux reduced the serial time of 5 days, 19 hours, and 13 minutes BLAST run to just a parallel time of 12 hours, 54 minutes. It was able to achieve a speedup of nearly 10.8.

1.5.2 mpiBLAST

mpiBLAST [mpibl] is an open-source parallelization of BLAST that achieves super-linear speed-up by segmenting a BLAST database and then having each node in a computational cluster search a unique portion of the database. Database segmentation permits each node to search a smaller portion of the database (one that fits entirely in memory), eliminating disk I/O and vastly improving BLAST performance. Because database segmentation does not create heavy communication demands, BLAST users can take advantage of low-cost and efficient Linux cluster architectures such as the bladed Beowulf.

mpiBLAST is a pair of programs that replace *formatdb* and *blastall* with versions that execute BLAST jobs in parallel on a cluster of computers with MPI installed. There are two primary advantages to using mpiBLAST versus traditional BLAST. First, mpiBLAST splits the database across each node in the cluster. Because each node's segment of the database is smaller it can usually reside in the buffer-cache, yielding a significant speedup due to the elimination of disk I/O. Also the data decomposition is done offline. The fragments reside in a shared storage as shown in Figure 1.5. Second, it allows BLAST users to take advantage of efficient, low-cost Beowulf clusters because the interprocessor communication demands are low.

The mpiBLAST algorithm consists of three steps:

1. segmenting and distributing the database, e.g., see Fig. 1.5,
2. running mpiBLAST queries on each node, and
3. merging the results from each node into a single output file.

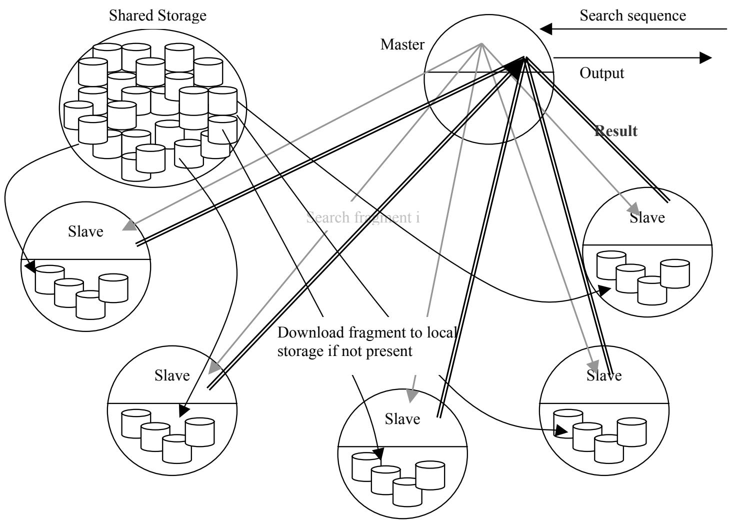

Fig. 1.5 Master-Slave worker model of mpiBLAST

The first step consists of a front-end node formatting the database via a wrapper around the standard NCBI *formatdb* called *mpiformatdb*. The *mpiformatdb* wrapper generates the appropriate command-line arguments to enable NCBI *formatdb* to format and divide the database into many small fragments of roughly equal size. When completed, the formatted database fragments are placed on shared storage. Next, each database fragment is distributed to a distinct worker node and queried by directly executing the BLAST algorithm as implemented in the NCBI development library. Finally, when each worker node completes searching on its fragment, it reports the results back to the front-end node who merges the results from each worker node and sorts them according to their score. Once all the results have been received, they are written to a user-specified output file using the BLAST output functions of the NCBI development library. This approach to generating merged results allows mpiBLAST to directly produce results in any format supported by NCBI's BLAST, including XML, HTML, tab-delimited text, and ASN.1.

The extra overhead incurred by the coordination and message passing may not pay off for small databases and small-to-medium length queries, but for databases that are too big to fit in the physical memory of a single node, it clearly offers an advantage.

1.5.3 Green Destiny

Green Destiny is a 240-processor supercomputer that operates at a peak rate of 240 gigflops [war02]. It fits in 6 square feet and consumes 3.2 kilowatts of power. mpiBLAST is benchmarked on the Green Destiny cluster by the Los Alamos National

Laboratory. Each node of this cluster consists of a 667 MHz Transmeta Crusoe TM5600 processor with 640 MB RAM and a 20 GB hard disk. Nodes are interconnected with switched 100Base-Tx ethernet. Each node runs Linux 2.4 operating system. It is able to achieve a speedup of over 160 for 128 processors. The database used is the GenBank nt of size 5.1 GB to run a query of predicted genes from bacterial geneome of size 300 KB. The cluster with one worker runs for about 22.4 hours whereas with 128 workers the query takes just 8 minutes. However, the scalability of this system is largely constrained by the time to merge results, which typically increase with the number of fragments.

1.6 HMMER- HIDDEN MARKOV MODELS

HMMer [edd98] is a statistical model, which is suited for many tasks in molecular biology, such as database searching and multiple sequence alignment of protein families and protein domains, although they have been mostly developed for speech recognition since the 1970s. Similar to the ones used in speech recognition, an HMM is used to model protein family such as globins and kinases. The most popular use of the HMM in molecular biology is as a probabilistic profile of a protein family, which is called profile HMM. From a family of proteins or DNA a profile HMM can be made for searching a database for other members of the family.

Internet sources for profile HMM and HMM-like software package are listed in the below table.

Software Tool	**Web Site**
HMMER	http://hmmer.wustl.edu
SAM	http://www.cse.ucsc.edu/research/compbio/sam.html
Pfam	http://pfam.wustl.edu
PFTOOLS	http://www.isrec.isb-sib.ch/ftp-server/pftools/
BLOCKS	http://blocks.fhcrc.org/
META-MEME	http://metameme.sdsc.edu
PSI-BLAST	http://www.ncbi.nlm.nih.gov/BLAST/

HMMER offers a more systematic approach to estimating model parameters. The HMMER is a dynamic kind of statistical profile. Like an ordinary profile, it is built by analyzing the distribution of amino acids in a training set of related proteins. However, an HMMER has a more complex topology than a profile. It can be visualized as a finite state machine, familiar to students of computer science. Finite state machines typically move through a series of states and produce some kind of output either when the machine has reached a particular state or when it is moving from state to state. The HMMER generates a protein sequence by emitting amino acids as it progresses through a series of states. Each state has a table of amino acid emission probabilities similar to those described in a profile model. There are also transition probabilities for moving from state to state.

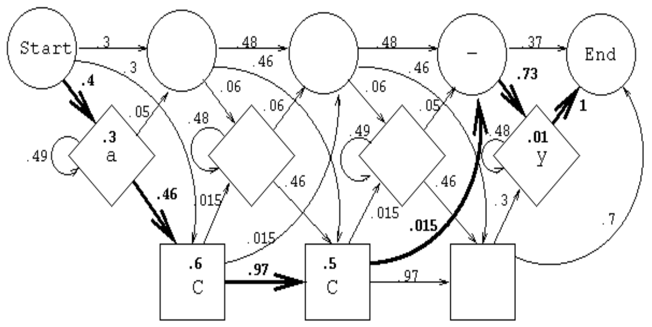

Fig. 1.6 A possible hidden Markov model for the protein ACCY. The protein is represented as a sequence of probabilities. The numbers in the boxes show the probability that an amino acid occurs in a particular state, and the numbers next to the directed arcs show probabilities which connect the states. The probability of ACCY is shown as a highlighted path through the model.

Figure 1.6 shows one topology for a hidden Markov model. Although other topologies are used, the one shown is very popular in protein sequence analysis. Note that there are three kinds of states represented by three different shapes. The squares are called match states, and the amino acids emitted from them form the conserved primary structure of a protein. These amino acids are the same as those in the common ancestor or, if not, are the result of substitutions. The diamond shapes are insert states and emit amino acids which result from insertions. The circles are special, silent states known as delete states and model deletions.

Transitions from state to state progress from left to right through the model, with the exception of the self-loops on the diamond insertion states. The self-loops allow deletions of any length to fit the model, regardless of the length of other sequences in the family. Any sequence can be represented by a path through the model. The probability of any sequence, given the model, is computed by multiplying the emission and transition probabilities along the path.

1.6.1 Scoring a Sequence with an HMMER

In Figure 1.6 a path through the model represented by ACCY is highlighted. In the interest of saving space, the full tables of emission probabilities are not shown. Only the probability of the emitted amino acid is given. For example, the probability of A being emitted in position 1 is 0.3, and the probability of C being emitted in position 2 is 0.6. The probability of ACCY along this path is

$$.4 * .3 * .46 * .6 * .97 * .5 * .015 * .73 *.01 * 1 = 1.76 \times 10^{-6}$$

As in the profile case described above, the calculation is simplified by transforming probabilities to logs so that addition can replace multiplication. The resulting number is the raw score of a sequence, given the HMMER. One common application of HMMER is classifying sequences in a database. The method is to build an HMMER with a training set of known members of the class, and then to compute the score of all sequences in the database. The resulting scores are ranked and a suitable threshold is selected to separate class members from the other sequences in the database.

1.6.2 Parallel Implementation of HMM

HMMer 2.2g provides a parallel *hmmpfam* program based on PVM (Parallel Virtual Machine), which is a widely used tool for searching one or more sequences against an HMM database and is provided by Dr. Eddy's Lab at the Washington University. In this implementation, the computation for one sequence is executed concurrently, the master node dynamically assigns one profile to a specific slave node for comparison. Upon finishing its job, the slave node reports the results to the master, which will respond by assigning a new profile. When all the comparison regarding this sequence is completed, the master node sorts and ranks all the results it collects, and outputs the top hits. Then the computation on the next sequence begins.

Using state-of-the-art multi-threading computing concept, some researchers [zhu03] implement a new parallel version of *hmmpfam* on EARTH (Efficient Architecture for Running Threads). EARTH is an event-driven fine-grain multi-threaded programming execution model, which supports fine-grain, non-preemptive fibers, developed by CAPSL (Computer Architecture and Parallel System Laboratory) at the University of Delaware. In its current implementations, the EARTH multi-threaded execution model is built with off-the-shelf microprocessors in a distributed memory environment. The EARTH runtime system (version 2.5) performs fiber scheduling, inter-node communication, inter-fiber synchronization, global memory management, dynamic load balancing and SMP node support. The EARTH architecture executes applications coded in Threaded-C, a multi-threaded extension of C.

For parallelizing *hmmpfam*, two different schemes are developed: one pre-determines job distribution on all computing nodes by a round-robin algorithm; the other takes advantage of the dynamic load balancing support of EARTH Runtime system, which simplifies the programmer's coding work by making the job distribution completely transparent. It shows a detailed analysis of the hmmpfam program and different parallel schemes, and some basic concepts regarding multi-threaded parallelization of HMM-pfam on EARTH RTS 2.5. When searching 250 sequences against a 585-family Hmmer database on 18 dual-CPU computing nodes, the PVM version gets absolute speedup of 18.50, while EARTH version gets 30.91, achieving a 40.1% improvement on execution time. On a cluster of 128 dual-CPU nodes, the execution time of a representative testbench is reduced from 15.9 hours to 4.3 minutes.

1.7 CLUSTALW

ClustalW is a tool used in computational biology to perform “multiple sequence alignment”. In practice, it means reading in a number of sequences representing sequences of biological data, calculating pairwise rankings of “alignedness” between them, forming a hypothetical tree of relationships between the sequences, and then performing adjustments to the sequences to make them all “align” with one another by introducing some gaps, etc. The following discussion involves basic steps involved in performing multiple sequence alignment. We then discuss how this is computationally intensive and discuss some parallel implementations using different approaches.

The basic alignment method was first devised by J D Thompson, Desmond Higgins, and Toby Gibson [tho94]. The basic multiple alignment algorithm consists of three main stages: 1) all pairs of sequences are aligned separately in order to calculate a distance matrix giving the divergence of each pair of sequences; 2) a guide tree is calculated from the distance matrix; 3) the sequences are progressively aligned according to the branching order (from tips to root) in the guide tree.

In the original CLUSTAL programs, the pairwise distances were calculated using a fast approximate method [bas87]. This allows very large numbers of sequences to be aligned, even on a microcomputer. The scores are calculated as the number of k-tuple matches (runs of identical residues, typically 1 or 2 long for proteins or 2 to 4 long for nucleotide sequences) in the best alignment between two sequences minus a fixed penalty for every gap. One can use either fast approximate method or the slower but more accurate scores from full dynamic programming alignments using two gap penalties (for opening or extending gaps) and a full amino acid weight matrix. These fast approximate methods virtually yield the same results as the exact methods as long as the sequences are not too dissimilar.

The guided tree which is used in the final multiple alignment process are calculated from the distance matrix of step 1 using the Neighbour-Joining method [sai87]. This produces unrooted trees with branch lengths proportional to estimated divergence along each branch. The root is placed by a “mid-point” method [th094-1] at a position where the means of the branch lengths on either side of the root are equal. These trees are also used to derive a weight for each sequence. The weights are dependent upon the distance from the root of the tree but sequences which have a common branch with other sequences share the weight derived from the shared branch. By contrast, in the normal progressive alignment algorithm, all sequences would be equally weighted.

Progressive alignment is the final stage in the ClustalW. This stage uses the series of pairwise alignments to align larger and larger groups of sequences, following the branching order in the guide tree. You proceed from the tips of the rooted tree towards the root. At each stage of the tree a full dynamic programming [amh67, th094-1] algorithm is used with a residue weight matrix and penalties for opening and extending gaps. It is appropriate to use dynamic programming here because the number of comparisons will be less compared to initial stage; thus leading to more accurate alignments. Each step consists of aligning two existing alignments or sequences. Gaps that are present in older alignments remain fixed. In the basic algorithm, new gaps that are introduced at each stage get full gap opening and extension penalties, even if they are introduced inside old gap positions. The average of all the pairwise weight matrix scores from the amino acids in the two sets of sequences is used in order to calculate the score between a position from one sequence or alignment and one from another,

If either set of sequences contains one or more gaps in one of the positions being considered, each gap versus a residue is scored as zero. The default amino acid weight matrices we use are rescored to have only positive values. Therefore, this treatment of gaps treats the score of a residue versus a gap as having the worst possible score. When sequences are weighted, each weight matrix value is multiplied by the weights from the 2 sequences.

If the total number of sequences is N, then the pairwise comparison step requires N*(N-1)/2 number of comparisons. N can be a very large number. Since at each step either fast approximation or full dynamic programming techiniques are used, this step will be more computationally intensive and throws a significant challenge for the researchers at improving the speed at this stage. Since the pairwise comparison state takes up most of the time and since it is easy to implement on multiprocessors, several parallel computing methods have been introduced. The next two sections will discuss about two very useful parallel programming approaches.

1.7.1 ClustalW-MPI

ClustalW-MPI is a distributed and parallel implementation of ClustalW [kuo03]. All three stages are parallelized in order to reduce the execution time. It uses a message-passing library called MPI and runs on distributed clusters as well as on traditional parallel computers. The first step in ClustalW is to calculate a distance matrix for N*(N-1)/2 pairs of sequences. This is an easy target for coarse-grained parallelization since all elements of the distance matrix are independent. The second step of ClustalW determines the guided tree (topology) of the progressive alignments. Finally the last step obtains the multiple alignments progressively. For the last two steps there is no simple coarse-grained parallel solution because of the data dependency between each stage in the guided tree.

The parallelization of the distance-matrix calculation is simply allocating the time-independent pairwise alignments to parallel processors. The scheduling strategy used in ClustalW-MPI is called fixed-size chunking [hag97]) where chunk of tasks are to be allocated to available processors. Allocating large chunk of sequences to available processors minimizes the communication overhead but may incur high processor idle time, whereas small batches reduce the idle time but may lead to high overhead.

Once we have the distance matrix, a guide tree needs to be produced to serve as the topology of the final progressive alignment. The algorithm for generating the guide tree is the neighbor-joining method (Saitou and Nei, 1987) [sai87]. Slight modifications were made so that the neighbor-joining tree can be done in O(n^2) time while still retain the same results as the original ClustalW. For the 500-sequence test data the tree generation takes about 0.04 % of the overall CPU time. In most cases the CPU time spent on this stage is less than 1% even for data containing 1000 sequences. ClustalW-MPI implementation parallelizes the searching of sequences having the highest divergence from all other sequences. A mixed fine- and coarse-grained approach is used for the final progressive alignment stage. It is coarse grained in that all external nodes in the guide tree are to be aligned in parallel. The efficiency obviously depends on the topology of the tree. For well-balanced guide tree, the ideal speedup can be estimated as N/ log N, where N is the number of nodes in the tree. Finally, the calculations of the forward and backward passes of the dynamic programming are also parallelized.

Experiments were conducted on test data comprising of 500 protein sequences with an average length of about 1100 amino acids. They were obtained from the BLASTP results with the query sequence (GI: 21431742), a cystic fibrosis transmembrane conductance regulator. Experiments were performed on a cluster that is made of eight dual-processor PCs (Pentium III, 800 MHz) and interconnected with the standard Fast Ethernet. The calculations of pairwise distances scale up as expected, up to 15.8 using 16 processors. For the essentially not parallelizable progressive alignment, this implementation shows that the speedup of 4.3 can be achieved using 16 processors.

From the above discussion it is evident that with the features of ClustalW-MPI, it is possible to speedup lengthy multiple alignments with relatively inexpensive PC clusters.

1.7.2 Parallel ClustalW, HT Clustalw and MULTICLUSTAL

Another parallel programming approach for ClustalW is undertaken by the SGI. This parallel version shows speedups of up to 10 when running ClustalW on 16 CPUs and significantly reduces the time requited for data analysis. The development of a high throughput version of ClustalW called HT ClustalW and the different methods of scheduling multiple MA jobs are discussed below. Finally the improvements of recently introduced MULTICLUSTAL algorithm and its efficient use for parallel ClustalW are discussed.

The basic ClustalW algorithm was parallelized by SGI using OpenMP [openmp] directives. Time profile analysis of the original Clustal W, using different numbers of G-protein coupled receptor (GPCR) proteins as inputs leads to the following discussion. Stages of the Clustal W algorithm: pairwise calculation (PW), guide tree calculation (GT), and progressive alignment (PA). Most of the time is spent in PW stage although the relative fraction is lower (50 %) for larger number of sequences compared to 90 % for smaller alignments. Therefore, the focus of parallization needs to be PW stage first. For larger number of sequences (>1000 and length of nucleotides 390) the time taken by the GT stage is also significant. For sequences greater than 1000 it is necessary to parallelize both PW and GT stages to get significant speedup.

PW Stage Optimization: As mentioned earlier, during the first stage N*(N-1)/2 pairwise comparisons have to be made to calculate the distance matrix. Because each comparison is independent of another, this part of the algorithm can be easily parallelized with the OpenMP "for" construct:

```
/* SGI pseudocode: Parallelism for pairwise
```

```
distance matrix calculation */

#pragma omp parallel private(i,j) {
#pragma omp for schedule(dynamic)

for(i=start;i<numseqs;i++)} {
    for(j=i+1;j<numseqs;j++)
        calc_pw_matrix_element();
    }
} /* End of pragma parallel */
```

Because inner “j”-loop varies, the OpenMP “dynamic” schedule is used in order to avoid load unbalance among different threads. In principle the “static” interleave schedule can be used here as well, but because each pairwise comparison takes varying amounts of time, the “dynamic” type works better. This implementation is only efficient on a shared memory system like the SGI Origin 3000 series. But even if this stage is parallelized, the scaling would still be limited to a low number of processors if no further optimization is done. For example, without parallelization second and third stages (GT and PA), the alignment of 1,000 GPCR protein sequences, where PW stage accounts for 50% of total time, would be only 1.8x faster when running on 8 CPUs according to Amdahl’s Law [amh67]. So, In order to achieve better scaling for larger, more compute-intensive alignments, efficient parallization techniques are needed for the GT and PA stages.

GT Stage Optimization: In the second stage (GT calculation), the most time-consuming part is determining the smallest matrix element corresponding to the next tree branch. This can be done in parallel by calculating and saving the minimum element of each row concurrently and then using the saved minimum row elements to find the minimum element of the entire matrix.

Experimental results have shown that the relative scaling of the parallel-optimized ClustalW for 100 and 600 GPCR sequences with the average length of 390 aminos (in terms of fraction of parallel code P [amh67]) is better for larger inputs since most of the time spent is in the first and second stages. For the larger inputs the time consumed by first and second stages is almost equal. Parallelization of these stages is more coarse grained, and as a result the OpenMP overhead becomes minimal compared to the finer grained parallelization of the third stage. The speedup of more than 10x is obtained for the MA of 600 GPCR proteins using 16 CPUs as compared to the one which was run on a single processor. Total time to solution is reduced from 1 hour, 7 minutes (single processor) to just over 6.5 minutes (on 16 CPUs of the SGI Origin 3000 series), and hence significantly increasing research productivity.

HT ClustalW Optimization: The need to calculate large numbers of multiple alignments of various sizes has become increasingly important in high-throughput (HT) research environments. To address this need, SGI has developed HT Clustal, basically, a wrapper program that launches multiple Clustal W jobs on multiple processors, where each Clustal W job is usually executed independently on a single processor.

In order to reproduce this High Throughput environment the following mix of heterogeneous MAs is constructed.

HT Clustal is used to calculate 100 different MAs for GPCR proteins (average length 390 amino acids). Each input file contains between 10 and 100 sequences taken randomly from a pool of 1,000 GPCR sequences. The number of sequences conforms to a Gaussian distribution with the average of 60 sequences and standard deviation of 20.

To optimize the throughput performance, the input sequences are pre-sorted based on a relative file size. The purpose of the pre-sorting is to minimize load unbalance and hence improve the scaling of HT Clustal. Experimental studies have shown that the improvement from pre-sorting becomes significant when the average number of jobs per CPU is on the order of 5. When the average number of jobs per CPU is greater than 5, it shows that the statistical averaging reduces the load unbalance and there is only minor improvement with pre-sorting.

With pre-sorting it is possible to achieve almost linear speedups. For the above example the speedups of 31x were achieved on 32 CPUs. For the larger test cases speedup of 116x was found on a 128-CPU SGI Origin 3000 series server, and hence reducing total time to solution from over 18.5 hours to just under 9.5 minutes. Because individual Clustal W jobs are processed on a single processor, HT Clustal can be used efficiently on both single system image SGI Origin 3000 series servers and distributed Linux clusters.

MULTICLUSTAL Optimization: The MULTICLUSTAL algorithm was introduced as an optimization of the ClustalW MA [yua99]. The MULTICLUSTAL alignment gives a domain structure, which is more consistent with the 3D structures of proteins involved in this alignment.

The algorithm searches for the best combination of ClustalW input parameters in order to produce more meaningful multiple sequence alignments (i.e. smaller number of gaps with more clustering). It does so by performing Clustal W calculations for various scoring matrices and gap penalties in the PW/GT and PA stages.

SGI has optimized the original MULTICLUSTAL algorithm by reusing the tree calculated in the PW/GT steps. Therefore, the guide tree is calculated only once for a given combination of PW parameters and is then used for all possible combinations of PA parameters. The performance (relative to that of original MULTICLUSTAL on 1 CPU) gives speedups range from 1.5 to 3.0 compared to the original algorithm running on the same number of processors. Similar time to solution can be obtained using the SGI modified MULTICLUSTAL on smaller number of CPUs compared to the original MULTICLUSTAL, thereby freeing additional computer resources without a performance degradation.

1.8 SPECIALIZED HARDWARE: FPGA

Over the past several years, key computational biology algorithms such as the Smith-Waterman have been implemented on FPGAs, and have enabled many computational

Table 1.1 Performance for different implementation.

Implementation	**Type**	**Processors per devices**	**Devices**	**Comparisons per seconds**
Celera	Alpha Cluster	1	800	250B
Paracel	ASIC	192	144	276B
Timelogic	FPGA	6	160	50B
StarBridge	FPGA	Unpublished	1	Unpublished
Splash 2	FPGA	14	272	43B
JBits(XCV1000)	FPGA	4,000	1	757B
JBits(XC2V6000)	FPGA	11,000	1	3,225B
Researcher from Japan(XCV2000E)	FPGA	144	1	4B

analysis that were previously impractical. Many problems in computational biology are inherently parallel, and benefit from concurrent computing models. There are several commercial systems currently available which all take different approaches. In academic area, many researchers proposed their implementations to address this problem. For a comparison, clusters of machines and custom VLSI systems are also included.

The first system listed in Table 1.1 is from Celera Genomics, Inc. Celera uses an 800 node Compaq Alpha cluster for their database searches. This arrangement is able to perform approximately 250 billion comparisons per second. The major advantage of such a multiprocessor system is its flexibility. The drawback, however, is the large cost associated with purchasing and maintaining such a large server farm.

The second system in the table is made by Paracel, Inc. Paracel takes a custom ASIC approach to do the sequence alignment. Their system uses 144 identical custom ASIC devices, each containing approximately 192 processing elements. This produces 276 billion comparisons per second, which is comparable to Celera's sever farm approach, but using significantly less hardware.

TimeLogic, Inc. also offers a commercial system but uses FPGAs and describes their system as using "reconfigurable computing" technology. They currently have six processing elements per FPGA device and support 160 devices in a system. This system performs approximately 50 billion matches per second. This is significantly lower in performance than the Celera or Paracel systems, but the use of FPGAs results in a more flexible system which does not incur the overheads of producing a custom ASIC.

StarBridge Systems [sta04] has developed a reconfigurable computing system using FPGAs that can deliver 10X to 100X or greater improvement in computational efficiency (compared to traditional RISC processor based machines) as per their white paper. Their system employs a dual processor motherboard and a single Hypercomputer board with nine Xilinx XC2V6000-BG1152 Virtex-II FPGAs and two XC2V4000-BG1152 Virtex-II FPGAs, yielding approximately 62 million gates per board. Details about processing elements are unpublished and the market price for this tailored system is relatively high.

Splash 2 [hoa93] is a legacy system loaded with 272 FPGAs, each supplying 14 processing elements, producing a match rate of 43 billion matches per second. These are respectable numbers for ten year old technology in a rapidly changing field.

The JBits [guc02] implementations using a Xilinx XCV1000 Virtex device implements 4,000 processing elements in a single device running at 188Mhz in the fully optimized version. This results in over 750 billion matches per second. And if the newer Virtex II family is used, a single XC2V6000 device can be used to implement approximately 11,000 processing elements. At a clock speed of over 280Mhz, this gives a matching rate of over 3.2 trillion elements per second.

Finally, the research group from Japan [yam02] also proposed their implementation using FPGAs. They use one PCI board with Xilinx XCV2000E FPGAs and implement 144 processing elements. The time for comparing a query sequence of 2048 elements with a database sequence of 64 million elements by Smith-Waterman algorithm is about 34 sec, which is about 330 times faster than a desktop computer with a 1Ghz Pentium III.

1.8.1 Implementation Details

The data dependencies from the Smith-Waterman algorithm indicate that calculations may proceed in parallel across the diagonals of the array. That is, if the *nxm* comparisons performed in the algorithm is viewed as a two dimensional array, then the algorithm can be seen as proceeding from the upper left corner of the array, where S0 is compared to T0, downward and to the right until the final value of d is computed using the comparison of Sn and Tm. There are two major implement approaches: multi-threaded computation and systolic array.

1.8.1.1 Multi-Threaded Computation The parallelism elements on each diagonal line are processed at once. Therefore, the order of the computation can be reduced to $m + n - 1$ from mn if m elements can be processed in parallel. If the size of the hardware is not large enough to compare m elements at once, the first p elements (suppose that the hardware process p elements in parallel) of the query sequence are compared with the database sequence at once, and the scores of all p^{th} elements are stored in temporal memory. Then, the next p elements of the query sequence are compared with the database sequence using the scores stored in the temporal memory.

Suppose that the length of the query sequence (m) is longer than the number of processing units on the FPGA (p). Then, in the naive approach:

1. The first p elements of the query sequence are compared and the intermediate results (all p^{th} scores on lower edge of upper half) are stored, and

2. Then, the next p elements of the query sequence are compared using the intermediate results.

In this case, it takes $2x2x(p + n - 1)$ cycles to compare the two sequences, and processing units become idle for one clock cycle in every two clock cycles as described above.

We can reduce the computation time by the multi-thread computation method. In the multi-threaded computation:

1. P elements on the diagonal line in upper half are processed, and the score of p^{th} element is stored on temporal registers, and

2. Then, the next p elements on the diagonal line in lower half are processed without waiting for one clock cycle using the intermediate result. By interleaving the processing of elements in upper half and lower half, we can eliminate the idle cycles of the processing elements. The clock cycles become $2x(p+n-1)+2xp$, which is almost equal to $2xn$ because n is much longer than p in most cases.

When the length of the query sequence (m) is longer than twice the number of the processing units $(2p)$, the multi-thread computation is repeated according to the length of the query sequence.

The advantage of this approach is that it can use the off-the-shelf boards from FPGA manufacturers. It is easy to obtain the boards with latest FPGAs (namely larger FPGAs) and the performance of the approach is almost proportional to the size of FPGAs. The disadvantage is that off-the-shelf FPGA boards do not have enough hardware resources for homology search. Especially memory size and memory bandwidth are not sufficient. Because of this limitation, query sequences can not be compared with long database sequences at once. Therefore, query sequences are always compared with subsequences of the database sequences (automatically divided during the search), and results against only the fragments in the subsequences can be shown. Research group from Japan takes this approach.

1.8.1.2 Systolic Sequence Comparison One property of the dynamic programming for computing edit distances is that each entry in the distance matrix depends on adjacent entries. This locality can be exploited to produce systolic algorithms in which communications is limited to adjacent processors. There are two mappings, both exploiting the locality of reference by computing the entries along each anti-diagonal in parallel [hoa92, lav96, lav98, lip85].

The first one is “bidirectional array” as shown in Figure 1.7. The source and target sequences are shifted in simultaneously from the left and right, respectively. Interleaved with the characters are the data values from the first row and column of the dynamic programming matrix. When two non-null characters enter a processor from opposite directions a comparison is performed. On the next clock tick the characters shift out and the values following them shift in. This processor now determines a new state based on the result of the comparison, the two values just shifted in, and its previous state value, using the same rule as in the dynamic programming algorithm. When the string are shifted out they carry with them the last two row and column of the dynamic programming matrix, and hence the answer. Comparing sequences of lengths m and n requires at least $2max(m + 1, n + 1)$ processors. The number of

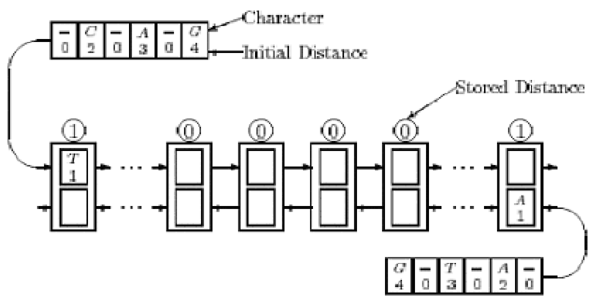

Fig. 1.7 Systolic array for sequence comparison.

steps required to compute the edit distance is proportional to the length of the array. With the bidirectional array, the source sequence must be cycled through the array once for each target sequence in the database. The source and target sequences are both limited in length to half of the array's length.

With respect to the shortcomings of the bidirectional array, "unidirectional array" process data in one direction. The source sequence is loaded once and stored in the array starting from the leftmost PE. The target sequences are streamed through the array one after another, separated by a control character. With the source sequence loaded and the target sequences streaming through, the array can achieve near 100% PE utilization. The length of the array determines the maximum length of the source sequence. The target sequence, however, can be of any length. Together, these properties make the unidirectional array more suitable and efficient than the bidirectional array for database searches. A unidirectional array of length n can compare a source sequence of length at most n and to a target sequence of length m in O$(n + m)$ steps.

Both bidirectional systolic array and unidirectional array have been implemented on the Splash 2 system using Xilinx FPGAs. The JBits implementation follows the same approach but use more advanced features of FPGAs.

1.8.1.3 Runtime Reconfiguration The JBits implementation explore the use of the run-time reconfiguration using Xilinx JBit toolkit. The JBits toolkit is a set of Java tools and APIs that permit direct impementation and reconfiguration of circuits for the Xilinx Virtex family of FPGAs. JBits was particularly useful in the implementation of this algorithm because there were several opportunities to take advantage of run-time circuit customization. In addition, the systolic approach to the computation permitted a single parameterizable core representing the processing element to be designed, then replicated as many times as necessary to implement the fully parallel array.

The logic implementation of the algorithm is shown in Fig. 1.8. Each gray box represents a LUT /flip-flop pair. This circuit demonstrates four different opportunities for run-time circuit customization. Three of these are the folding of the constants for the insertion, deletion and substitution penalties into the LUTs. Rather than explicitly feeding a constant into an adder circuit, the constant can be embedded in the circuit,

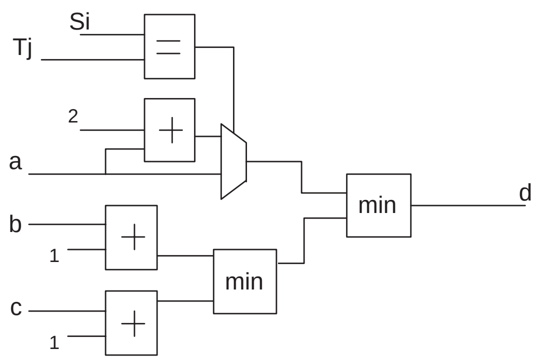

Fig. 1.8 The combinational logic of the Smith-Watermann circuit.

resulting in (in effect) a customized constant adder circuit. Note that these constants can be set at run time and may be parameters to the circuit.

The fourth run-time optimization is the folding of the match elements into the circuit. In genomic databases, a four character alphabet is used to represent the four bases in the DNA molecule. These characters are typically denoted A, T, G, C. In this circuit, each character can be encoded with two bits. The circuit used to match Si and Tj does not require that both strings be stored as data elements. In this implementation, the S string is folded into the circuit as a run-time customization. Note that unlike the previous optimizations, the string values are not fixed constants and will vary from one run to another. This means that the entire string S is used as a run-time parameter to produce the customized circuit.

This design uses a feature of the algorithm first noted by Lipton and Lopresti [lip85]. For the commonly used constants, 1 for insert/delete and 2 for substitution, b and c can only differ from a by +1 or -1, and d can only differ from a by either 0 or 2. Because of this modulo 4 encoding can be used, thus requiring only 2 bits to represent each value. The final output edit distance is calculated by using an up-down counter at the end of the systolic array. For each step, the counter decrements if the previous output value is one less than the current one and it increments otherwise. The up-down counter is initialized to the match string length which makes zero the minimum value for a perfect match.

1.9 CONCLUSION

Relying on Moore's law alone for performance demands of computational biology applications may prove detrimental. An amalgamation of better architectures, clever algorithms, computation system with higher raw CPU performance; with less power consumption and higher bandwidth will be required to meet demands of computational biology. We do not expect the general purpose computers to provide the cost-performance for most of these problems. We expect architectural changes to support computational biology.

REFERENCES

[aar03] Aaron E. Darling, Lucas Carey, Wu-chun Feng, "The Design, Implementation, and Evaluation of mpiBLAST", *ClusterWorld Conference & Expo in conjunction with the 4th International Conference on Linux Clusters: The HPC Revolution 2003*, June 2003.

[alt90] Altschul, S.F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. "Basic local alignment search tool", *J. Mol. Biol.*,215,403-410, 1990.

[alt96] Altschul SF, Gish W., "Local alignment statistics", *Methods Enzymol.*, 266:460-80 1996.

[alt97] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", *Nucleic Acids Res.*, 25(17): 3389-3402, 1997.

[amh67] Amdahl, G., "Validity of the single-processor approach to achieving large-scale Computing capabilities." *Proc. AFIPS Conf.*, 1967.

[bas87] Bashford, D., Chothia, C. and Lesk, A.M. "Determinants of Protein Fold: Unique features of Globin Amino Acid Sequences.",*J. Mol. Biol.*, 196, 199-216.1987

[bjo02] Bjornson, R.D., Sherman, A.H., Weston, S.B., et. al. "TurboBLAST: A Parallel Implementation of BLAST Built on the TurboHub", *International Parallel and Distributed Processing Symposium: IPDPS Workshops*, p.0183, 2002.

[che03] J. Cheetham, F. Dehne, S. Pitre, A. Rau-Chaplin, and P. J. Taillon, "Parallel CLUSTAL W For PC Clusters", *International Conference on Computational Sciences and Its Applications*, 2003.

[edd98] S.R. Eddy, "Profile hidden Markov models", *Bioinformatics"*, 1998.

[got82] Gotoh, O., "An improved algorithm for matching biological sequences", *J. Mol. Biol.*, 162, 705-708, 1982.

[guc02] Guccione, Steven A., Eric Keller, "Gene Matching Using JBits", *Field-Programmable Logic and Applications*, Springer-Verlag, Berlin, 2002.

[hag97] Hagerup,T.,"Allocating independent tasks to parallel processors: an experimental study.", *J. Parallel Distrib. Comput.*, 47, 185-197,1997.

[hoa92] Hoang, Dzung T., "FPGA Implementation of Systolic Sequence Alignment", *International Workshop on Field Programmable Logic and Applications, Vienna, Austria*, Aug. 31 - Sept. 2, 1992.

[hoa93] Hoang, Dzung T., "Searching Genetic Databases on Splash 2",*IEEE Workshop on FPGAs for Custom Computing Machines"*, pp. 185–191, IEEE Computer Society Press, 1993.

[hug96] Hughey, R.,"Parallel hardware for sequence comparison and alignment", *Comput. Appl. Biosci.*, 12, 473-479, 1996.

[jan03] Chintalapati Janaki and Rajendra R. Joshi, "Accelerating comparative genomics using parallel computing", *Silicon Biology 3*, 0036, Bioinformation Systems, 2003.

[kar90] Karlin, S., and Altschul, S. F., "Method for assessing the statistical significance of molecular sequence features by using general scoring schemes", *Proceedings of the National Academy of Science*, USA 87, 2264-2268, 1990.

[kuo03] Kuo-Bin Li, "ClustalW-MPI: ClustalW analysis using distributed and parallel computing". *Bioinformatics*,2003.

[lav96] Lavenier, Dominique, "SAMBA: Systolic Accelerators for Molecular Biological Applications", *IRISA Report*, March 1996.

[lav98] Lavenier, Dominique, "Speeding up genome computations with a systolic accelerator", *SIAM News*, vol. 31, no. 8, Oct. 1998.

[lip85] Lipton, Richard J., Daniel Lopresti, "A Systolic Array for Rapid String Comparison", *Chapel Hill conference on VLSI, Computer Science Press*, pp. 363-376, 1985.

[men04] Xiandong Meng, Vipin Chaudhary, "Bio-Sequence Analysis with Cradle's 3SoC™ Software Scalable System on Chip",*SAC '04*, March 14-17, 2004.

[mpibl] ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools/, NCBI Toolbox download site

[NCBI] NCBI, http://www.ncbi.nlm.nih.gov

[openmp] *www.openmp.org*.

[osw01] Oswaldo Trelles.,"On the Parallelization of Bioinformatic Applications", *Briefings in Bioinformatics*, Vol.2, May 2001.

[parac] Paracel, "PARACEL-BLAST: Accelerated BLAST software optimized for Linux clusters."

[pea90] W. R. Pearson, "Rapid and sensitive sequence comparison with FASTP and FASTA", *Methods Enzymol.* 183: 63-98, 1990.

[rog00] Rognes, T. and SeeBerg E., "Six-fold speedup of Smith-Waterman sequence database searches using parallel processing on common microprocessors", *Bioinformatics*, Vol.16, no. 8, 699-706, 2000.

[roy89] S. Roy and V. Chaudhary, "Design Issues for a High-Performance Distributed Shared Memory on Symmetrical Multiprocessor Clusters", Cluster Computing: The Journal of Networks, Software Tools and Applications, pp. 177 - 186, 2 (1999) 3 1999.

[sai87] Saitou,N. and Nei,M. "The neighbor-joining method: a new method for reconstructing phylogenetic trees.",*Mol. Biol. Evol.*,4, 406-425,1987.

[sha01] Ilya Sharapov, "Computational Applications for Life Sciences on Sun Platforms: Performance Overview", Whitepaper, 2001

[smw81] Smith, T.F. and Waterman, M.S.,"Identification of common molecular subsequences", *J. Mol. Biol.*, 147, 195-197, 1981.

[sta04] Starbridge Systems, www.starbridgesystems.com

[tho94] Thompson, J.D., Higgins, D.G. and Gibson, T.J.,"CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice.",*Nucleic Acids Research*, 22:4673-4680. 1994.

[th094-1] , Thompson, J.D., Higgins and Gibson," Improved sensitivity of profile searches through the use of sequence weights and gap excision.",*CABIOS* ,10, 19-29, 1994.

[tre01] O. Trelles, "On the parallelization of bioinformatics application, *Brief Bioinform.*, 2(2): 181-194, 2001.

[war02] M. Warren, E. Weigle, W. Feng, "High-Density Computing: A 240-Node Beowulf in One Cubic Meter," *Proceedings of SC2002*, November 2002.

[wun70] Needleman, S. and, Wunsch C., "A general method applicable to the search for similarities in the amino acid sequence of two sequences", *J. Mol. Biol.*, 48(3), 443-453, 1970.

[yam02] Y. Yamaguchi, T. Maruyama, and A. Konagaya, "High Speed Homology Search with FPGAs", *Pacific Symposium on Biocomputing 7*,271-282, 2002.

[yua99] Yuan, J., Amend, A.,Borkowski,j.,DeMarco, R., Bauiley, W., Liy,Y., Xie, G., and Blevins, R. "MULTICLUSTAL: a systematic method for surveying Clustal W alignment parameters. Bioinformatics", 15(10), 862, 1999.

[zha00] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A greedy algorithm for aligning DNA sequences.", *J Comput Biol.*, 7(1-2): 203-214, 2000.

[zhu03] Weirong Zhu, Yanwei Niu, Jizhu Lu and Guang R. Gao, "Implementing Parallel Hmm-pfam on the EARTH multithreaded Architecture",*Proceedings of the computational Systems Bioinformatics*,2003.