On the Parallelization of Bioinformatic Applications

Oswaldo Trelles

Computer Architecture Department, University of Malaga

29071 Malaga, Spain, Phone +(34) 52132823, Fax (34)952132790, e-mail: ots@ac.uma.es

Abstract

This document surveys the computational strategies followed to parallelize the most used software in the bioinformatics arena. The studied algorithms are computationally expensive and their computational patterns range from regular, such as database searching applications, to very irregularly structured patterns (phylogenetic trees). Fine- and coarse-grained parallel strategies are discussed for these very diverse sets of applications. This overview outlines computational issues related to parallelism, physical machine models, parallel programming approaches, and scheduling strategies for a broad range of computer architectures. In particular, it deals with shared, distributed, and shared/distributed memory architectures.

Keywords: Parallel Computers, Parallel Software, Bioinformatics

Introduction

➧ *Information overload*

With the growth of the information culture, efficient digital searches are needed to extract and abstract useful information from massive data. In the biological and biomedical fields, massive data take the form of bio-sequences flat files, 3D structures, motifs, 3D microscopic image files, and more recently, videos, movies, animations, etc. However, while genome projects and DNA arrays technology are constantly and exponentially increasing the amount of data available (for statistics see http://www3.ebi.ac-.uk/Services/DBStats), our ability to absorb and process this information remains near constant.

It was only few years ago when we were confident that *evolution* in computer-processing speed, increasing exponentially like some areas of knowledge in molecular biology, could handle the growing demand posed by bioinformatic applications. Processing power has jumped from the once-impressive 4.77 MHz in the early Intel 8088 to more than one GHz current frequencies in the AMD-K7 and Pentium III gallery. Most probably when reading this document the Pentium IV and AMD-K8, with up to 1.5 GHz, will be available (for details, http://www.prism.uvsq.fr/mirror/CIC/summary/local). This exponential growth rate can also be observed in the development of practically every computer component, such as the number of CPU transistors, memory access time, cache size, etc.

However, contemporary genome projects have delivered a blow to this early confidence. From the completion of the first whole organism's genome (saccharomyces, mid-1998), the growth rates for biological data have become a detriment to sequential computing processing capability. At this point, sequential (one-processor) computing can allow only a small part of the massive, multidimensional biological information to be processed. Under this scenario, comprehension of the data and understanding of the data-described biological processes could remain incomplete, causing us to lose vast quantities of valuable information because CPU-power and time constraints could fail to follow critical events and trends.

➧ *Computational resources*

From a computational point of view, there are several ways to address the lack of hard computing power for bioinformatics. The first is by developing new, faster *heuristic* algorithms that reduce computational space for the most time-consuming tasks (Altschul, *et al.*, 1997, Pearson and Lipman, 1988). The second is incorporating these algorithms into the ROM of a specialized chip (*i.e.,* the bio-accelerator at Weizmann Institute, http://sgbcd//weizmann.ac.il/).

The third and most promising consideration, however, is parallel computing. Two or more microprocessors can be used simultaneously, in parallel processing, to divide and conquer tasks that would overwhelm a single, sequential processor. However promising, parallel computing still requires new paradigms in order to harness the additional processing power for bioinformatics.

Before this document embarks on a detailed overview of the parallel computing software currently available to biologists, it is useful to explore a few general concepts about computer architectures, as well as the parallel programming approaches that have been used to address bioinformatic applications.

Parallel Computers

➔ *Parallel Computer architectures: Taxonomy.*

A *parallel computer* uses a set of processors that are able to cooperate in solving computational problems (Foster, 1994). This co-operation is made possible, first, by splitting the computational load of the problem (tasks or data) into parts and, second, by reconnecting the partial computations in order to create an accurate outcome. The way in which load distribution and reconnection (communications) are managed is heavily influenced by the system that will support the execution of a parallel application program.

Parallel computer systems are broadly classified into two main models based on Flynn's (1972) specifications: single-instruction multiple-data (SIMD) machines, and multiple-instruction multiple-data MIMD machines.

SIMD machines are the dinosaurs of the parallel computing world; once powerful, but now facing extinction. A typical SIMD machine consists of many simple processors (hundreds or even thousands), each with a small local memory. Every processor must execute, at each computing or ‘clock’ cycle, the same instruction over different data. When a processor needs data stored on another processor, an explicit communication must pass between them to bring it to local memory. The complexity and often the inflexibility of SIMD machines, strongly dependent on the synchronization requirements, have restricted their use mostly to special-purpose applications.

MIMD machines are more amenable to bioinformatics. In MIMD machines, each computational process executes at its own rhythm in an *asynchronous* fashion with complete independence of the other computational processes (Hwang and Xu, 1998). Memory architecture has a strong influence on the global architecture of MIMD machines, becoming a key issue for parallel execution, and frequently determines the optimal programming model.

It is really not difficult to distinguish between shared and distributed memory. A system is said to have *shared-memory architecture* if any process, running in any processor, has direct access to any local or remote memory in the whole system. Otherwise, the system has distributed memory architecture.

Shared memory architecture brings several advantages to bioinformatic applications. For instance, a single address map simplifies the design of parallel programs. In addition, there is no 'time penalty' for communication between processes, because every byte of memory is accessible in the same amount of time from any CPU (uniform memory access, UMA architecture). However, nothing is perfect, and shared memory does not scale well as the number of processors in the computer increases.

Distributed memory systems scale very well, on the other hand, but the lack of a single physical address map for memory incurs a time penalty for inter-process communication (non-uniform memory access, NUMA architecture).

Current trends in multiprocessor design try to achieve the best of both memory architectures. A certain amount of memory physically attaches to each node (distributed architecture), but the hardware creates the image of a single memory for the whole system (shared architecture). In this way, the memory installed in any node can be accessed from any other node as if all memory were local with only a slight time penalty.

A few years ago, two technological breakthroughs made possible another exciting approach to parallel computing. The availability of very fast processors in workstations, together with the widespread utilization of networks, led to the notion of a "virtual parallel computer" that connected several fast microcomputers by means of a Local Area Network. This distributed-memory system was called *multi-*

computer architecture.

Multi-computer configurations are constructed mainly with clusters of workstations (COWs), although one emerging multi-computer architecture is beowulf-clusters (http://www.beowulf.org), which are composed of ordinary hardware components (like any PC) together with public domain software (like Linux, PVM or MPI). A server node controls the whole cluster, serving files to the client nodes.

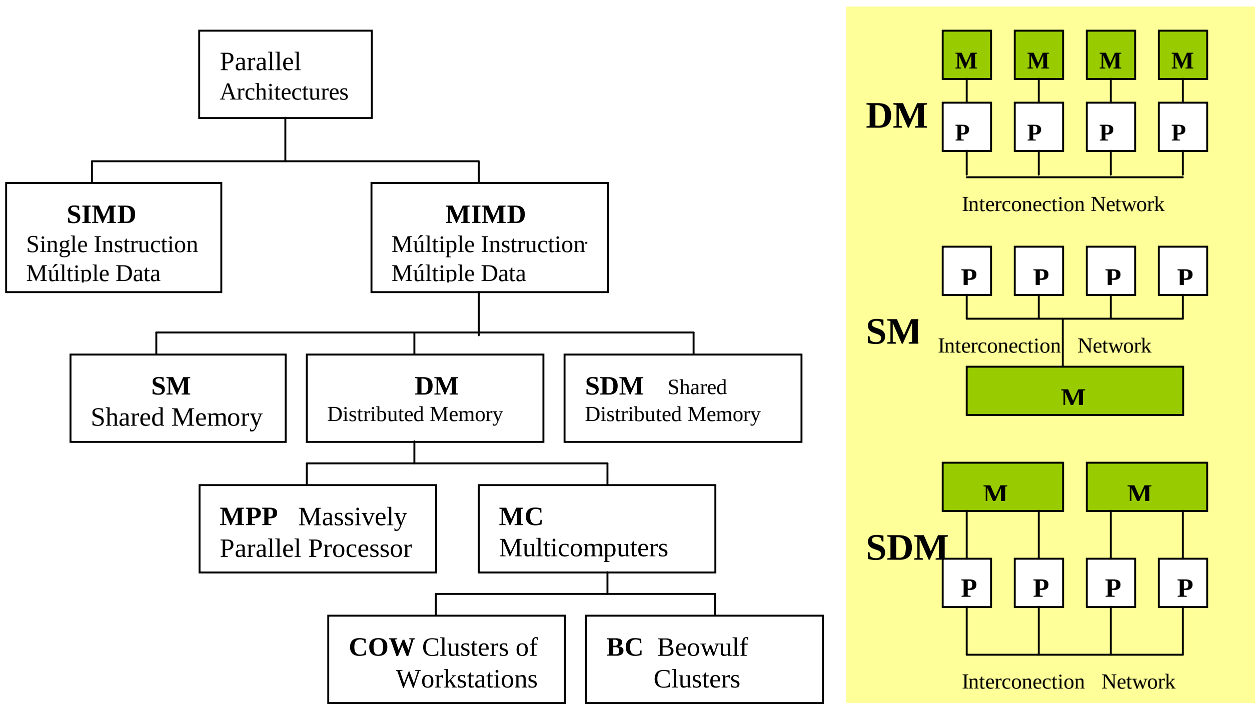


Figure 1. Summarized Parallel Computer Architecture Taxonomy and Memory Models. Many forms of parallelism exist today. Some architectures bring together a relatively small number of very tightly-coupled processors. In other designs, the coupling of processors is relatively loose, but the number of processors can scale up to the thousands. A diagram of parallel architecture taxonomy is presented on the left. On the right are the most used memory models available for these architectural designs.

Multi-computers bring several advantages to parallel computing: cost (on average, one order of magnitude cheaper for the same computational power), maintenance (replacing fault nodes), scalability (adding new nodes), and code-portability. Some drawbacks also exist, such as the lack of available software that enables management of the cluster as one integrated machine. In addition to this, current network technology has high latency and insufficient bandwidth to handle fast parallel processing. These factors limit the effectiveness of this architecture at the present time, although it looks promising given the expected capabilities of future technologies.

Parallel Programming Models

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. The method used to divide tasks and rejoin the end result can be used as a point of reference to compare different alternative models for parallel programs.

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

Most of today's advanced single microprocessor architectures are based on the Superscalar and Multiple

Issue paradigms (MIPS-R10000, Power-PC, Ultra-Sparc, Alpha 21264, Pentium III, etc.) These paradigms have been developed to exploit Instruction Level Parallelism (ILP): the hardware level of granularity.

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 (or an automatic *analyzer*) 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 real implementations, however, the performance of a parallel program is decreased by synchronization between processes, interaction (information interchanges), and load imbalance (idle processors while others are busy). Co-ordination between processes represents sources of *overhead*, in the sense that they 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 interactions 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, a crucial question arises: how will the parallel processes interact to coordinate the behaviour of each other? Communications are needed to enforce correct behavior and create an accurate outcome.

➧ *Communications*

When shared memory is available, interprocess communication is usually performed through shared variables. When several processes are working over the same logical address space, locks, semaphores or *critical sections* (blocks of code that only one process can execute at a time) are required for safe access to shared variables.

When the processors use distributed memory, all interprocess communication must be performed by sending messages over the network. With this message-passing paradigm, the programmer needs to keep in mind *where* the data is, *what* to communicate, and *when* to communicate to *whom*. Library subroutines are available to facilitate message-passing constructions: PVM (Sunderam, 1990), MPI (http://www.mpi-forum.org/index.html), etc. As one might imagine, writing parallel code for a disjointed memory space address 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 (*i.e.*, TreadMarks, http://www.cs.rice.edu/~willy/TreadMarks/overview.html).

➧ *Task scheduling strategies*

Common knowledge gained from working on parallel applications suggests that obtaining an efficient parallel implementation is fundamental to achieve 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.

The simple task scheduling strategy is based on a master/slave approach. In essence, one of the processors acts as a *master*, scheduling and dispatching blocks of tasks (*e.g.,* pairwise sequence alignments) to the slaves which, in turn, perform the typical calculations specified by the algorithm. When the slave completes one block, the master schedules a new block of tasks and repeats this process until all tasks have been computed. Efficiency can be improved by slaves pre-fetching tasks from the master so as to overlap computations and communications. Efficiency is further improved by catching problems in slaves, so that slaves communicate with the manager only when no problems are available locally.

As the number of slaves scales upward, slaves can be divided into sets, each with a sub-master, in a hierarchical fashion. Finally, in a fully decentralized model, each processor manages its own pool of tasks, and idle slave processors request tasks from other processors. One can easily see how bioinformatics applications, with their massive data calculation loads, would be amenable to parallel processing.

At this point, a very schematic and abbreviated description of parallel architectures has been presented for easier comprehension. A more academic, up-to-date, and detailed description can be found, for example, in Tanenbaum 1999, (chapter 8: Parallel Computer Architectures).

BioInformatic Applications

In this section, different and routinely used algorithms, will be presented to describe the strategies followed to parallelize bioinformatic software. The discourse has been organized by the task-level computational pattern observed in such algorithms, from regular to irregular structured (Rodriguez *et al.* 1998). Traditionally, a regular-irregular classification, also named synchronous/asynchronous (and their respective semi-regular and loosely synchronous levels), has been used in such a way that it was closely related to the characteristic that computations were performed over dense or sparse matrices. However, when working with non-numerical applications, as is the case for most of bioinformatic applications, the rate of free-dependent tasks, the data access pattern, and the task homogeneity, are appropriate indices used to classify applications.

➧ *Regular computational pattern: Database searching*

Database searching (*DBsrch*) is the most heavily used bioinformatic application. It is also one of the most familiar applications to begin a discussion about parallelization in bioinformatics: *DBsrch* 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.

The primary influx of information for bioinformatics applications 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. Results often suggest functional, structural, or evolutionary analogies between the sequences.

Two main sets of algorithms are used for pairwise comparison (the individual task in a *DBsrch* application): (a) *exhaustive* algorithms based on dynamic programming methodology (Needleman and Wunsch, 1970; Smith and Waterman, 1981); and (b) *heuristic* (faster and most used) approaches such as the FASTA (Wilbur and Lipman, 1983, Lipman and Pearson 1985, Pearson W.R. and Lipman 1988) and BLAST (Altschul, *et al.* 1990, 1997) families.

DBsrch applications allow two different granularity alternatives to be considered: fine- and coarse-grained parallelism. Early approaches focused on data-parallel over SIMD machines (notably the ICL-DAP massive parallel computer) starting with the pioneering work of Coulson *et al.* (1987).

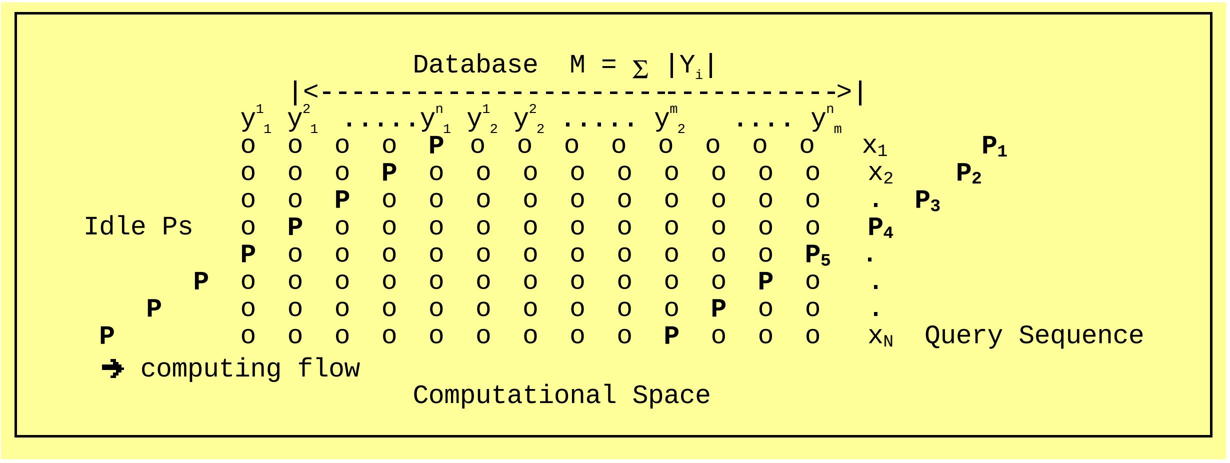
Deshpande *et al.* (1991) and Jones (1992) presented a work on hypercubes and CM-2 computers. Soon after, Sturrock and Collins (1993) implemented the exhaustive dynamic programming algorithm of Smith and Waterman (1981) in the MasPar family of parallel machines (from the minimum 1024 processors configuration of MP-1 systems up to a 16384 processors MP-2 systems). They roughed out one of the first remote servers over parallel machines (the BLITZ server at the EMBL, http//:www.embl-heidelberg.de) that is still active at the EBI (http://www.ebi.ac.uk/MPsrch/).

Simple and elegant dynamic programming-based algorithms compute an $S_{N,M}$ matrix (N and M being the sequence lengths). The $S_{i,j}$ cell is defined by the expression:

$$S_{i,j} = max\ [\ \{\ S_{i-1,j-1} + w(x_i,y_j)\ \},\ \{\ S_{i-1,j} + \alpha_g\ \},\ \{\ S_{i,j-1} + \alpha_g\}]$$

where w represents a scoring scheme for every pair of residues x_i,y_j, and α_g is a negative value representing the penalty for introducing or extending a gap of length g. To compute the $S_{i,j}$ cell, data dependencies exist with the value of the previous cell in the same diagonal, and the best values are on the left of the previous row and on top of the previous columns.

Fine-grain means, in this case, that processors will work together in computing the S matrix, cell by cell. Edmiston and Wagner (1987), and Lander *et al.* (1988) organized the CM-2 machine as an array of processors to compute in diagonal-sweep fashion the matrix S (see Figure 2). An advantage is that this strategy only requires local communications (in each step, P_i sends $S_{i,j}$ to P_{i+1} to allow it to compute $S_{i+1,j}$ in the next step, while P_i computes $S_{i,j+1}$). Query sequence length determines the maximum number of processors able to be assigned, and processors remain idle at begin/end steps. Both inconveniences are important due to the high number of processors usually present in SIMD architectures.



```
                          Database  M = ∑ |Yi|
                     |<-------------------------------->|
                  y¹₁ y²₁ .....yⁿ₁ y¹₂ y²₂ ..... yᵐ₂   .... yⁿₘ
                  o  o  o  o  P  o  o  o  o  o  o  o  o   x₁      P₁
                  o  o  o  P  o  o  o  o  o  o  o  o  o   x₂     P₂
                  o  o  P  o  o  o  o  o  o  o  o  o  o   .   P₃
    Idle Ps       o  P  o  o  o  o  o  o  o  o  o  o  o   P₄
                  P  o  o  o  o  o  o  o  o  o  o  o  P₅  .
               P  o  o  o  o  o  o  o  o  o  o  o  P  o   .
            P     o  o  o  o  o  o  o  o  o  o  P  o  o   .
         P        o  o  o  o  o  o  o  o  o  P  o  o  o   xN  Query Sequence
     ➔ computing flow
                          Computational Space
```

Figure 2. Diagonal-sweep fine-grained workload distribution for SIMD machines to avoid data dependencies. Rows are distributed along processors (residue x_i of query sequence is assigned to processor P_i) and processor P_i starts its computations with a delay of i columns. There will be (P x (P-1)) idle processors at the beginning and at the end of computations.

Around this time, Collins *et al.* (1987) proposed a *row-sweep* workload distribution, splitting the sequence database into groups of 4096 residues to be assigned to a 64x64 array of processors. This modification was significant, because solving both problems (number of P's greater than sequence length and idle processors), addresses an important problem: data-dependencies. In fact, in step j, P_i computes only partially the cell $S_{i,j}$ ($S_{i-1,j-1}$ is received by a message from P_{i-1} in step $j-1$ and the best column value is in the same processor). At this point, the best horizontal value is needed to complete the cell final value. To broadcast this value, only $\log_2 P$ messages are used when processor P sends a message in iteration i to processor $P+2^i$ (with i=1...12). Note that a given processor needs to send a message only when it changes its best row value (a very unlikely event); thus, in practical terms, the number of messages is much lower.

It might seem unnecessary that the last two paragraphs have been used to discuss parallel strategies for computers that, to use a colloquial expression, are in *danger of extinction*. However, apart from its historical interest, there are other good reasons. As we can see right away, a coarse-grained approach is best for a great number of tasks (such as most of today's parallel bioinformatic problems). However, several other applications exist for which there are not enough independent tasks to be solved concurrently. It is still possible to learn from early approaches, and obtain fruitful conclusions that improve new parallel solutions.

There are several proposed strategies for achieving coarse-grained parallelism in *DBsrch* applications. Most of them can be explained on the basis of the following general pseudo-code:

```
[1]   Get-Parameters, get Query-Sequence and Perform Initializations
[2]   while (there is sequences in the Database) {
[3]           fetch a DB-sequence
[4]           score = Algorithm(Query-Sequence, DB-sequence, parameters)
[5]           Maintain a trace of the best results (DB-seq-Code, score)
      }
[6]   Results Optimization
[7]   Report Best Results
```

Table 1: General sequential pseudo-code for a DBsrch application. Step [1] sets the initial stage of the algorithm, and Step [2] manages the algorithm extension which works until the number of database sequences is exhausted. Step [3] loads the next sequence to be compared against the query sequence in step [4]. The result value is often used to rank the best results as in step [5], and finally, specific implementations can incorporate a last optimization step (i.e., assessing the statistical significance of results) described as step [6] and report results in step [7].

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, we need to keep in mind, 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 two sequences (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 (Deshpande, *et al.* 1991).

Additional improvements are obtained by applying buffering strategies that reduce or eliminate slave inactivity while waiting for a new message (server data starvation). The master processor can send, at the outset, more than one block of sequences to each slave, so that a slave has a new block at the ready to continue working as soon as each block is completed. (Trelles *et al.* 1994a).

Several methods have been used to determine the size of the block of sequences to be distributed. The simplest way is to divide the database in *n* chunks (*n* being the number of slave processes) and obviously assign one chunk to each slave (Martino, *et al.* 1994) The data chunks can even reside in a local disk storage. To minimize load unbalancing, sequences are ordered by size and are assigned in round-robin fashion to chunks. The strategy is simple, inexpensive, and effective.

Unfortunately, it also presents at least two difficult problems: (a) to perform the distribution it is necessary to know in advance the number of processors (*n*); and (b) when working in heterogeneous environments, such as multi-computers clusters of workstations, the CPU-time needed to process each chunk can be quite different, depending on the CPU-power and the CPU availability in each node.

A direct solution divides the database in *m* blocks of sequences ($m >> n$) of fixed length (with block size around 4-16 Kbytes, aiming to maximize the network bandwidth) and assigns blocks to slaves on demand. In this way, the maximum imbalance at the end of computations is proportional to the block size, and scheduling cost (including message-passing) is proportional to *m*. The major scheduling-distribution cost is normally shadowed by using buffering strategies, as explained above. An additional specialization can be obtained by using blocks of variable size (Trelles *et al.* 1994b).

This last approach allows a pattern of growing-size/decreasing-size messages with a minimal scheduling cost. It is especially suitable for clusters of workstations because it avoids server data starvation due to scheduling latencies. If the first blocks are short, the first servers may finish computing before a new block of data is available to them. If the first blocks are large, the last slaves must wait a substantial amount of time for their first block of data to be dispatched. Moreover, large blocks in the last steps of the data distribution may increase overall processing time due to poor load balancing.

For distributed memory parallel machines, the blocks of sequences arrive at slaves via message passing from a master that deals with the file system. It is also possible that the master sends to slaves only a pointer in the database, and the slaves load the sequences by themselves through the NFS (Network File System) or another particular element, *i.e.,* the CFS (Concurrent File System).

When shared memory is available, a counter-variable, which serves as a pointer into the database, manages the workload distribution. Since the counter is located in the shared memory, each processor can access it in a guarded region, obtain the value, and move the pointer to the next block. This type of system has been implemented for the Cray Y-MP (Jülich, 1995).

Two simple notes complete this epigraph:

(a) The *Achilles heel* of message passing is the relatively limited data transmission bandwidth in the communication pathway. In these architectures, the communication/computation ratio must be low to efficiently port algorithms. It will always be harder to parallelise Fasta or Blast than a dynamic programming algorithm; and

(b) when there are several query sequences for database searching (*i.e.,* in the case of a *DBsrch* server) a process-level of granularity can be applied (in fact, this approach is used at the NCBI (http://www.ncbi.nlm.nih.gov/BLAST)

However, there is a more important thing to be learned at this point. When more tasks than processors are available, the simplest and most effective strategy is coarse-grained parallelization. This is so fundamental that presenting a new algorithm with this feature goes together with its parallel coarse-grained implementation. Some good examples are:

Structural biology (electron microscopy): determines viral assembly mechanisms and identifies individual proteins. The computational-intensive task in this algorithm is associated with imaging the 3D structure of viruses from electron micrographs (2D projections). The number of tasks is related to the set of candidate orientations for each particle, such calculations being at different orientations, completely independent of each other.

Protein structure prediction: this task involves searching through a large number of possible structures representing different energy states. One of the most computationally intensive tasks calculates the solvent accessible surface area that can be measured on individual atoms if the location of neighboring atoms is known.

Searching 3D structure databases: as the number of protein structures known in atomic detail increases, the demand for searching by similar structures also grows. A new generation of computer algorithms has been developed for searching by: (a) extending dynamic programming algorithms (Orengo *et al.* 1992); (b) importing strategies from computer vision areas (Fisher *et al.* 1992); (c) using intra-molecular geometrical information, as distances, to describe protein structures (Holm and Sander, 1993, 1994); and (d) finding potential alignments based on octomeric C alpha structure fragments, and determining the best path between these fragments using a final dynamic programming step followed by least squares superposition (Shindyalov and Bourne, 1998).

Linkage analysis: genetic linkage analysis is a statistical technique used for rapid and largely automated construction of genetics maps from gene linkage data. One key application of linkage analysis aims to map human genes and locate disease genes. The basic computational goal in genetic linkage analysis is to compute the probability that a recombination occurs between two loci L_1 and L_2. Most frequently used programs estimate this *recombination function* by using a maximum likelihood approach (Ott, 1991).

All of the previous examples fit perfectly to coarse-grained parallel applications, due to the large number of *independent* tasks and the regular computational pattern they exhibit, together with the low communication/computation rate they present. All these features make them suitable for parallelism with high efficiency rates. However, several other interesting examples have non-regular computational patterns, and they need particular strategies to better exploit parallelism.

Let's take a deeper look into the last example. In the parallelization of LINKMAP, Miller et al. (1992) first used a machine-independent parallel programming language known as Linda. It was compared to the use of machine-specific calls on the study case of a Hypercube computer and a network of workstations, concluding that a machine independent code could be developed using that tool with only a modest sacrifice in efficiency. One particular hypothesis says there are many pedigrees and/or many candidate θ vectors, treating each likelihood evaluation for one pedigree as a separate task. If there are enough tasks, a good load balancing can be obtained. Godia *et al.* (1992) use a similar strategy for the MENDEL program. However, Gupta *et al.* (1995) observe that typical optimization problems have a dimension of only 2 or 3, thus, there is no need for a large number of processors. In conclusion, it is important to integrate parallelization strategies for individual function evaluation (coarse grained), with a strategy to parallelize the gradient estimation (fine-grained).

- *Semi-Regular Computational patterns*

A similar problem arises in the parallelization of hierarchical multiple sequence alignments, *MSA* (Corpet, 1988; Gotoh 1993; Miller 1993, Thompson *et al.* 1994). The first steps for solving an MSA include calculating a *cross similarity matrix* between each pair of sequences, followed by determining the alignment topology and finally solving the alignment of sequences, or clusters themselves.

Pairwise calculation provides a natural target for parallelization because all elements of the distance matrix are independent (for a set of *n* sequences *n(n-1)/2* pairwise comparisons are required). Computing the topology of the alignment (the order in which the sequences will be grouped) is a relatively inexpensive task, but solving the clustering (guided by the topology) is not that amenable to parallelism. This is due to the fact that, at this stage, many tasks are to be solved (for a set of *n* sequences it is necessary to solve *n-1* alignments). However, only those tasks corresponding to the external nodes of the topology can be solved concurrently.

Certainly, parallel strategies for the *cross matrix* calculation have been proposed (Gonnet *et al.* 1992; Date *et al.* 1993, SGI^{TM}, 1999), all of them in a coarse-grained approach. In addition, when the MSA is embedded in a more general clustering procedure (Trelles *et al.* 1998), combining a dynamic planning strategy with the assignment of priorities to the different types of active tasks using the principles of data locality has allowed us both to exploit the inherent parallelism of the complete applications, and to obtain performances that are very close to optimal.

However, at present and strictly speaking, the last step in MSA remains unsolved for parallel machines. When the work is carried out following a coarse-grained parallelization scheme for distributed memory architectures, it is then necessary to exchange the sequences and/or clusters that are being modified due to the insertion of gaps during their alignment, which is extremely expensive.

For this, we should look back and learn from the earliest fine-grained parallel solutions applied to sequence comparison. Today, when mixed shared/distributed memory architectures are available, this could be an excellent exercise that -it should be stressed- is far from being an academic one. A full solution probably should combine a coarse-grained solution when computing the cross similarity matrix with a fine-grained solution for solving the topology. Many challenges are yet to be overcome.

- *Irregular computational patterns*

Applications with irregular computational patterns are the hardest to deal with in the parallel arena. In numeric computation, irregularity is mostly related to sparse computational spaces which introduce hard problems for data parallel distributions (fine-grained approaches) and data dependencies. The latter reduces the number of independent tasks, which affords little chance to develop efficient coarse-grained parallel implementations.

A good example of this comes from another routine task in biological sequence analysis; that of building phylogenetic trees (Gribskov, M. and Devereux, J. 1991, Cavalli-Sforza and Edwards, 1967; Felsenstein, 1973, 1988). Earlier approaches to apply maximum likelihood methods for very large sets of sequences have been centered in the development of new simpler algorithms, such as the *fastDNAml* (Olsen *et al.*, 1994), which has been ported to parallel architectures using the P4 package (Butler & Lusk 1992). A current parallel version of the fastDNAml, implemented in C with communications under MPI is available at http://www.santafe.edu/~btk/science-paper/bette.html and at the Pasteur Institute under TreadMarks http://www.cs.rice.edu/~willy/TreadMarks/overview.html. Even these simplified approaches have been known to be quite computationally intensive. In fact, they were reported to have consumed most of the CPU time of the first IBM SP1 installation in the Argonne National Laboratory (1993).

Let's centre our attention on the original Felsenstein version of the method (implemented in the PHYLIP package, available at evolution.genetics.washington.edu). In very simple terms, the maximum likelihood method searches for a tree and a branch length that have the greatest probability of being produced from the current sequences that form that tree. The algorithm proceeds by adding sequences into a given tree topology in such a way that maximizes the likelihood topology (suitable for coarse-grained parallelism). Once the new sequence is inserted, a local-optimization step is performed to look for minor rearrangements that could lead to a higher likelihood.

These rearrangements can move any sub-tree to a neighbouring branch. Given a current tree T_k with likelihood L_k, one of its *k* nodes is removed and rearranged in its two neighbour nodes which produce two new trees, T_{k1} y T_{k2}, with likelihood L_{k1} and L_{k2}, respectively. The tree with greater likelihood value (including L_k) is chosen as the new best-tree and it replaces the current-tree. This procedure is performed until the set of nodes to rearrange is exhausted, as can be observed in Table 2.

```
Current-best-tree $T_k$ ($L_k$) [from insertion step]
for i = 1 to n-tasks
    Remove sub-tree i from $T_k$ and produce $T_{k1}$ and $T_{k2}$
    Likelihood evaluation for $T_{k1}$ and $T_{k2}$ ($L_{k1}$ and $L_{k2}$)
    Current-best-tree $T_k$ = tree with greater likelihood ($T_k$, $T_{k1}$, $T_{k2}$)
    end for
```

Table 2 Pseudo-code for local-optimization step in DNAml algorithm

Strictly speaking, only those nodes without leaves and a depth of at least 2 (not hanging from the root node) can be reorganized, which represent *2k-6* tasks (*k* being the number of species). For a large number of sequences, it could be addressed in a coarse-grained parallel solution by distributing the n-tasks among different processors.

Unfortunately, the reorganization task of one node is dependent on the reorganization of the previous nodes, due to the replacement of the *current best-tree*. In fact, each new-optimization task must be performed over the last *best-tree* found, and not over the initial topology. This leaves only two tasks (likelihood evaluation for T_{k1} and T_{k2} topologies) that can be solved in parallel. In other words, the maximum theoretical speed-up of this step will be limited to this value (2), independent of the number of processors used in the computation.

There is no generic procedure to address this type of irregular problem; hence, a good initial approach includes a detailed analysis of the computational behavior of the algorithm. In this specific case, a careful *run-time* analysis of the algorithm shows that the number of times a tree with a likelihood better than the current likelihood is obtained is extremely low (see Figure 4). From this behaviour, it is possible to conclude that the probability of a *new current-best-tree* event is rather low; or conversely, in most of the cases there is a high probability that a best tree will not be produced. The most important implication of this *run-time* observation is that, having evaluated the probability of rearranging a node, the next likelihood evaluation can be started with the same tree used to evaluate the previous one. In this way, the task dependencies in the local optimization step are avoided (Ceron *et al.* 1998, Trelles *et al.* 1998).

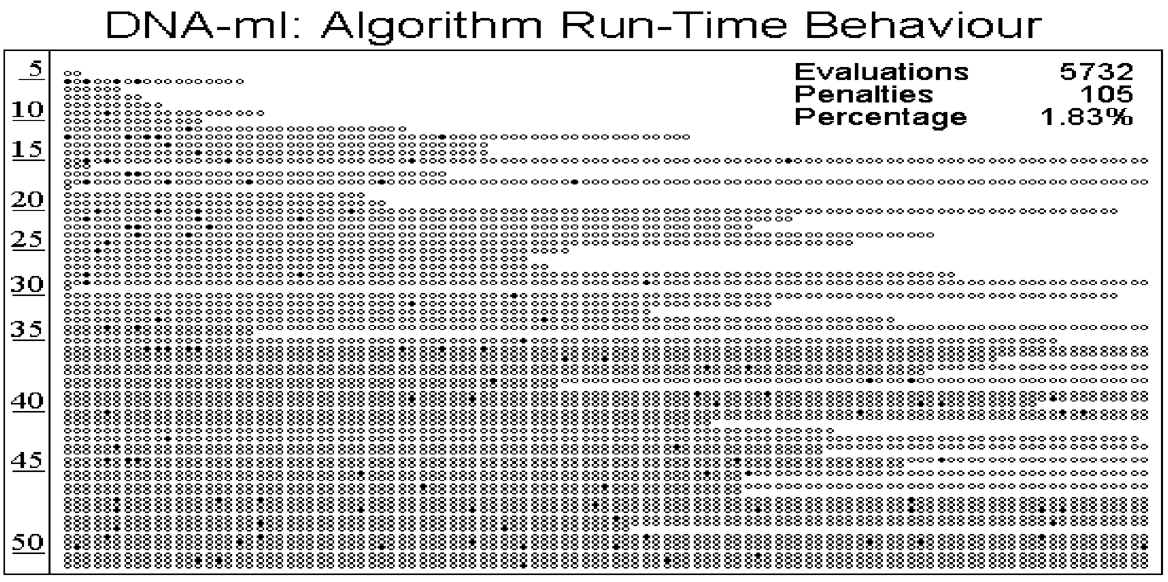


Figure 4. Example of the run-time behaviour of the DNAml algorithm in the optimization step (4) using 50 sequences. The number of circles in each horizontal line represents the number of optimization tasks successively performed as a function of the number of sequences already incorporated into the topology (in the left-hand column). Filled circles show those points in which a new maximum value was detected. At the top right-hand corner of the figure, the total number of tree likelihood evaluations performed by Ceron *et al.* algorithm is presented, together with the number of extra-evaluations (parallel-penalty) incurred by the algorithm and the very low penalty evaluation percentage.

Conclusions

➧ *Inter-disciplinary work*

Born almost at the same time 50 years ago, molecular biology and computer science have grown explosively as separate disciplines. However, just as two complementary DNA strands bind together in a double helix to better transmit genetic information, an evolving convergence has created an interrelationship between these two branches of science. In several areas, the presence of one without the other is unthinkable.

Not only has traditional sequential Von Neumann-based computing been fertilized through this interchange of programs, sequences, and structures, but the biology field has also challenged high-performance computing with a broad spectrum of demanding applications (for CPU, main memory, storage capacity, and I/O response time). Strategies using parallel computers are driving new solutions that seemed unaffordable only few years ago.

➧ *reusing available software*

Parallel computing has shown itself to be an effective way to deal with some of the hardest problems in bioinformatics. The use of parallel computing schemes expands resources to the size of the problem that can be tackled, and there is already a broad gallery of parallel examples from which we can learn and import strategies, allowing the development of new approaches to challenges awaiting solution, without the need to 're-invent the wheel'.

Today, it should be natural to "think in parallel" when writing software, and it should be natural to exploit the implicit parallelism of most applications when more than one processor is available. In most bioinformatic applications, due to a high number of independent tasks, the simplest approaches are often the most effective. These applications scale better in parallel, are the least expensive, and are the most portable among different parallel architectures.

➧ *new challenges*

However, several other challenges in bioinformatics remain unsolved as far as parallel computing is concerned. They represent attractive challenges for biologists and computer scientists in the years ahead.

Acknowledgements

I would like to thank Manolo Ujaldon and Charlie Killin for valuable comments on this document. This work has been partially supported by grant 1FD97-0372 from the EU-FEDER Programme (Fondos Europeos de Desarrollo Regional), and a DIGICYT-CSIC grant for the promotion of University-Research council associated units between the National Center for Biotechnology and the Computer Architecture Department, at the University of Malaga, Spain.

References

Altschul, S.F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. (1990), "Basic local alignment search tool", J.Mol.Biol. 215. 403-410
Altschul, S.F., Madden T.L., Schaffer A.A., Zhang J., Zhang Z., Miller W. and Lipman D.J., (1997) "Gapped BLAST and PSI-BLAST: A new Generation of Protein DB search Programs", Nucleid Acids Research (1997) v.25, n.17 3389-3402
Argonne National Laboratory, (1993), "Early experiences with the IBM SP1 and the High Performance Switch", (Internal report ANL-93/41)
Butler, R. and Lusk, E. (1992), "User's guide to the P4 programming system, (Argonne National

Laboratory Technical report TM-ANL-92/17)

Cavalli-Sforza, L.L. and Edwards, A.W.F. (1967), "Phylogenetic analysis: models and estimation procedures", Am. J. Hum. Genet. 19: 233-257.

Ceron, C., Dopazo, J., Zapata, E.L., Carazo, J.M. and Trelles, O. (1998), "Parallel Implementation for DNAml Program on Message-Passing Architectures", Parallel Computing and Applications, 24 (5-6), 701-716

Coulson, A.F.W., Collins, J.F. and Lyall, A., (1987) "Protein and Nucleid Acid sequence database searching: a suitable case for Parallel Processing", Computer J., (39), 420-424

Corpet, F. (1988), "Multiple sequence alignments with hierarchical clustering", Nucleic Acid Research, (16), 10881-10890

Date, S, Kulkarni,R., Kulkarni, B., Kulkarni-kale, U. and Kolaskar,A. (1993), "Multiple alignment of sequences on parallel computers". CABIOS (9)4, 397-402

Deshpande, A.S., Richards, D.S. and Pearson, W.R. (1991), "A platform for biological sequence comparison on parallel computers", CABIOS (7), 237-247

Edmiston, E. and Wagner, R.A. (1987) "Parallelization of the dynamic programming algorithm for comparison of sequences", Proc. of 1987 International Conference on Parallel Processing pp.78-80

Felsenstein, J. (1973), "Maximum-likelihood estimation of evolutionary trees from Continuous Characters". Society of Human Genetics 25: 471-492.

Felsenstein, J. (1988), "Phylogenies from molecular sequences: inference and reliability". Annu. Rev. Genet. 22: 521-565.

Jones, R., (1992) "Sequence pattern matching on a massively parallel computer", CABIOS (8), 377-383

Jülich A., (1995), "Implementations of BLAST for parallel Computers", CABIOS 11, 1 (3-6)

Flynn, M.J., (1972), "Some Computer Organizations and their Effectiveness", IEEE Trans.on Computers, vol.C-21 (948-960)

Fisher, D., Bachar, O., Nussinov, R. and Wolfson, H. (1992), "An efficient automated computer vision, based technique for detection of three-dimensional structural motifs in proteins", J.Biomol.Struct.Dyn. 9:769-789

Foster, I. (1994), "Designing and Building parallel programs: concepts and tools for parallel software engineering", Addison-Wesley Publishing Company, Inc.
(On-line version: http://wotug.ukc.ac.uk/parallel/books/addison-wesley/dbpp/)

Gribskov, M. and Devereux, J. (1991),"Sequence Analysis Primer", UWBC Biotechnical Resource Series

Godia, T.M., Lange, K., Miller, P.L. and Nadkarni, P.M., (1992), "Fast computation of genetic likelihoods on human pedigree data", Human Heredity, 42:42-62

Gonnet, G.H., Cohen, M.A. and Benner, S.A. (1992) "Exhaustive matching of the entire protein sequence database". Science, (256), 1443-1445

Gotoh, O., (1993), "Optimal alignment between groups of sequences and its application to multiple sequence alignment". CABIOS (9), 2, 361-370

Gupta, S.K., Schäffer, A.A., Cox, A.L., Dwarkadas, S. and Zwaenepoel, W. (1995), "Integrating parallelization strategies for linkage analysis", Computers and Biomedical Research, (28) 116-139

Holm, L. and Sander Ch. (1993), "Protein structure comparison by alignment of distance matrices", J.Mol.Biol. 233:123-138

Holm, L. and Sander Ch. (1994), "Searching protein structure databases has come of age", Proteins 19:165-173

Hwang Kai and Xu Zhiwei (1998), "Scalable Parallel Computing: Technology, Architecture, Programming", McGraw-Hill Series in Computer Engineering

Korber, B., Muldoon, M., Theiler, J., Gao, F., Gupta, R., Lapedes, A., Hahn, B., Wolinsky, S., Bhattacharya, T. (2000), "Timing the Ancestor of the HIV-1 Pandemic Strains", Science 288 pp.1757-1-759.

Lander, E., Mesirov,J.P. and Taylor W. (1988) "Protein sequence comparison on a data parallel computer". Proc. of 1988 International conference on Parallel Processing pp.257-263

Lipman, D.J. and Pearson, W.R. (1985), "Rapid and sensitive protein similarity searches", Science, 227, 1435-1441

Martino, R.L., Johnson, C.A., Suh, E.B., Trus, B.L. and Yap, T.K. (1994) "Parallel computing in Biomedical research". Science (256) 902-908

Miller, W. (1993) "Building multiple alignments from pairwise alignments". CABIOS (9) 2, 169-176.

Needleman, S.B. and Wunsch, C.D. (1970), "A general method applicable to the search for similarities in the aminoacid sequence of two proteins", J.Mol.Biol., 48, 443-453

Olsen, G.J., Matsuda, H., Hagstrom, R. and Overbeek, R. (1994), "fastDNAml: a tool for construction of philogenetic trees of DNA sequences using maximum likelihood, *CABIOS* 10 41-48

Orengo, C.A., Brown, N.P. and Taylor, W.T. (1992), "Fast stucture alignment for protein databank searching", Proteins, 14:139-167

Ott, J., (1991) "Analysis of Human Genetic Linkage", The Johns Hopkins University Press, Baltimore and London (Revised Edition).

Pearson W.R. and Lipman D.J.; (1988), "Improved tools for biological sequence comparison", Proc.Natl.Acad.Sci. USA (85), 2444-2448

Rodriguez, A., Fraga, L.G. de la, Zapata, E.L., Carazo, J.M. and Trelles, O., (1998), “Biological Sequence Analysis on Distributed-Shared Memory Multiprocessors”, 6th Euromicro Workshop on Parallel and Distributed Processing. Madrid, Spain.

Shindyalov, I.N, and Bourne, P.E. (1998), "Protein structure alignment by incremental combinatorial extension (CE) of the optimal path", Protein Engineering 11 (9) 739-747

SGI^{TM}, (1999), "SGI Bioinformatics performance report" at http://www.sgi.com/chembio

Smith, T.F. and Waterman, M.S. (1981), "Identification of common molecular subsequences", J.Mol.Biol, 147, 195-197

Sturrock, S.S. and Collins, J., (1993), "MPsrch version 1.3", BioComputing Research Unit, University of Edinburgh, UK.

Sunderam, V., Manchek, R., Dongarra, J., Geist, A., Beguelin, A. and Jiang, W. (1993), "PVM 3.0 User`s Guide and Reference manual". Oak Ridge National Laboratory.

Tanenbaum, A., (1999), "Structured Computer Organization", Ed. Prentice-Hall, Fourth Edition.

Thompson, J.D., Higgins, D.G. and Gibson, T.J, (1994), "Clustal W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting position-specific gap penalties and weight matrix choice", Nucleic Acids Research 22:4673-4680

Trelles,O., Zapata, E.L. and Carazo, J.M., (1994a), "Mapping strategies for sequential sequence comparison algorithms on LAN-based message passing architectures, In *Lecture Notes in Computer Science*, vol 796; High Performance Computing and Networking, Springer-Verlag, Berlin, 197-202

Trelles, O., Zapata, E.L. and Carazo, J.M., (1994b), "On an efficient parallelization of exhaustive sequence comparison algorithms on message passing architectures, CABIOS 10 (5), 509-511

Trelles, O., Andrade, M.A., Valencia, A., Zapata E.L. and Carazo, J.M., (1998), "Computational Space Reduction and Parallelization of a new Clustering Approach for Large Groups of Sequences", BioInformatics vol.14 no.5 (pp.439-451)

Trelles, O., Ceron, C., Wang, H.C., Dopazo, J. and Carazo, J.M., (1998), "New phylogenetic venues opened by a novel implementation of the DNAml Algorithm", BioInformatics vol.14 no.6 (pp.544-545)

Wilbur, W.J. and Lipman, D.J, (1983), "Rapid similarity searches in nucleic acid and protein databanks", Proc.Natl.Acad.Sci. USA, 80. 726-730