LISP AND SYMBOLIC COMPUTATION, 7, 231–247 (1994)

Note: this paper is slightly older than the version published in LASC (only the formatting has changed).

Using Multilisp for Solving Constraint Satisfaction Problems: an Application to Nucleic Acid 3D Structure Determination

MARC FEELEY feeley@iro.umontreal.ca
MARCEL TURCOTTE turcotte@iro.umontreal.ca
GUY LAPALME lapalme@iro.umontreal.ca
Département d'informatique et de recherche opérationnelle, Université de Montréal, Montréal (Québec) Canada, H3C 3J7

Abstract. This paper describes and evaluates a parallel program for determining the three-dimensional structure of nucleic acids. A parallel constraint satisfaction algorithm is used to search a discrete space of shapes. Using two realistic data sets, we compare a previous sequential version of the program written in Miranda to the new sequential and parallel versions written in C, Scheme, and Multilisp, and explain how these new versions were designed to attain good absolute performance. Critical issues were the performance of floating-point operations, garbage collection, load balancing, and contention for shared data. We found that speedup was dependent on the data set. For the first data set, nearly linear speedup was observed for up to 64 processors whereas for the second the speedup was limited to a factor of 16.

Keywords: Parallel Computation, Symbolic Computation, Multilisp, Constraint Satisfaction, Functional Programming, Applications

1. Introduction

The work described here is part of an ongoing project on the determination of the three-dimensional structure of nucleic acids. Interest in nucleic acids has been fueled by the recent discovery of their catalytic activity. The detailed knowledge of the structure of nucleic acids is considered a crucial prerequisite to the comprehension and eventual manipulation of their function.

For a very large number of nucleic acids, the *sequence* of nucleotides (the chemical composition) is known but not the three-dimensional shape. This is due in part to the great progress in sequencing techniques and, in a related way, to mega sequencing projects, such as the Human Genome Project [8]. There is thus a great need for sequence analysis tools that infer shape from sequence data.

Most successful approaches to the structure determination problem rely on homology and computer graphics modeling. These techniques are motivated by the observation that natural selection has produced families of molecules in which the sequence of nucleotides has diverged widely, but the three-dimensional structure and the function have remained the same. The methods use nucleic acids of known structure from which pieces are extracted that have good sequence homology with regions of the target sequence. With the help of molecular display programs the global structure is manually constructed from these pieces. However, structure has been determined for few nucleic acids and thus this approach is limited. A review of RNA modeling techniques is presented in [9].

Our approach combines symbolic and numerical computation, as depicted in fig. 1. A "symbolic generation" step takes experimental and theoretical data as input and generates a preliminary pool of structures using the *Constraint Satisfaction Problem* (CSP) algorithm described in this paper. A "numerical" step processes this pool of structures using commercially available energy minimization and molecular dynamic packages. This two step approach has the virtue of reducing the size of the search space explored by the energy minimization method. The precision lost in the symbolic generation model is recovered in the numerical step. In this paper we discuss the symbolic generation step. Details of the numerical step can be found in [11]. A sequential version of the system, called MC-SYM for "Macromolecular Conformation by SYMbolic generation" [13], is in use in over 30 sites around the world, including several academic research centers and two pharmaceutical companies.

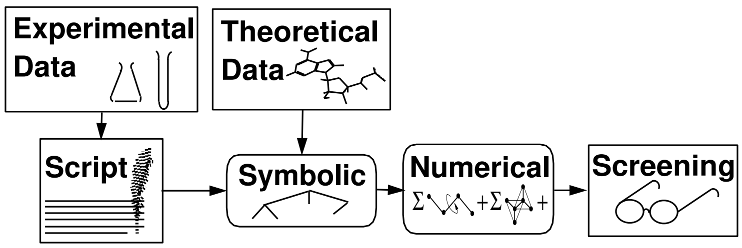


Figure 1. Flow of information and processes.

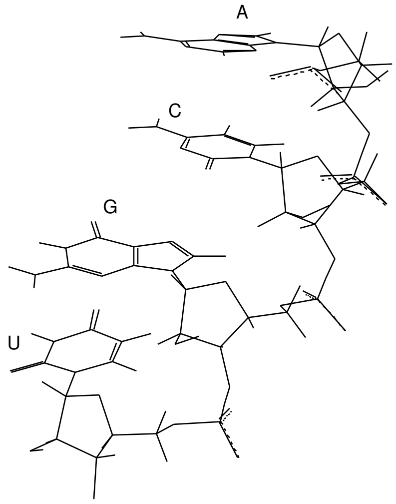
The next section introduces some background theory, and following sections explain the details of our method.

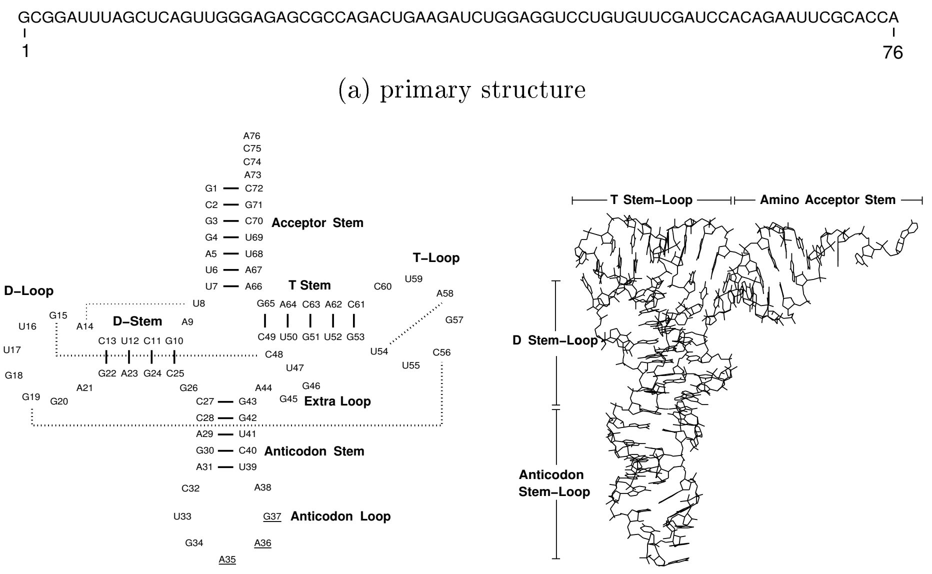
2. Background Theory: Nucleic Acid Structure

There are two types of nucleic acids (see [15] for a review); one is deoxyribonucleic acid (DNA), which carries the genetic information, and the other is ribonucleic acid (RNA) which serves as an intermediary in the protein synthesis but also may have catalytic properties.

The nucleic acids are chains of smaller molecules, the nucleotides (fig. 2). There are four types of nucleotides, A, C, G, and T for DNA and A, C, G, and U for RNA. The chains range from small nuclear RNAs, called snRNAs of less than 30 nucleotides, to large ribosomal RNAs, containing over 3000 nucleotides. The sequence of nucleotides is called the *primary structure*. The second level of organization, the *secondary structure*, arises from the well known fact that nucleotide bases can interact (base-pairing) and form double-helical domains. Finally, bases from single-stranded regions can interact in space and further fold the molecule; this defines the relative placement of double-helical domains and the exact 3D coordinates of all atoms, the *tertiary structure*.

Figure 2. The picture to the right shows the four types of ribonucleotides; from top to bottom A,C,G and U. "... nucleotides contain a phosphate group linked to a five-carbon-atom sugar group, which, in turn, is joined to a flat aromatic molecule that can be either a double-ringed purine or a single-ringed pyrimidine. Since they contain the sugar deoxyribose, the nucleotides of DNA are called deoxy-ribonucleotides, while those of RNA, which contain the sugar ribose, are known as ribonucleotides" [16].





(b) secondary structure (c) tertiary structure

Figure 3. The three levels of organization of nucleic acids. Figure (a) shows the linear sequence of nucleotides. Figure (b) shows the two-dimensional folding of the molecule: it shows the juxtaposition in space of distant nucleotides in the sequence (solid lines represent regular base-pairings while dotted lines show long-range tertiary interactions). Figure (c) shows the tertiary structure. The point of this complicated figure is not the details, but only to show that the tertiary structure is the actual 3D structure, including the location of all atoms in the nucleic acid.

The only biologically active RNA class for which tertiary structure has been determined is that of the transfer RNA molecules (tRNA), which are involved in the transcription of genetic code (DNA) to protein. The tRNA molecules are generally 75 nucleotides long and are composed of around 2000 atoms. Because tRNA molecules are the only nucleic acids of known structure they are also the benchmarks for modeling techniques and they will serve as examples in the remainder of the paper. Figure 3 shows the three levels of organization of the *yeast* Phenylalanine tRNA (entry number 1TRA of the Protein Data Bank [2]).

Researchers have developed reliable methods for determining the primary structure of proteins and nucleic acids. Those data are collected and made available by research organizations such as the *National Center for Biotechnology Information*; the latest release (81.0, 15 February 1994) of the *NCBI-GenBank Flat File* contains roughly $170,000,000$ bases from over $160,000$ reported sequences. On the other hand, determining the 3D structure by purely experimental means is still a time consuming task. This explains why the January 1994 Protein Data Bank release contains only 2428 three-dimensional structures from proteins, DNAs and RNAs (less than 1% of the known sequences).

Thus, one of the most important unsolved problems in molecular biology is still the *structure determination problem*: given a sequence of nucleotides, determine the three-dimensional structure of the biologically active molecule. But it may not be possible, at least in the short term, to solve this problem without additional information.

Additional information on the three-dimensional structure is provided by the method of comparative sequence analysis and by enzymatic and chemical methods. Comparative sequence analysis is based on the observation that corresponding RNA molecules from different organisms adopt a similar set of base-pairings, *i.e.* the molecules have a common secondary structure. By comparing the nucleotide sequences of RNA molecules it is possible to infer almost all secondary structure interactions and some tertiary interactions (see Appendix A.1 for more detail).

The use of specific enzymes, e.g. enzymes that cut single-stranded regions, and sequencing techniques provide additional information about the secondary structure of the molecule. Some chemical agents that are specific to the nucleotide bases can be used to detect paired and non-paired nucleotides.

Thus the structure determination problem can be reformulated as: given a sequence of nucleotides and a set of secondary and tertiary interactions, predict the three-dimensional structure of the molecule. The information from the comparative sequence analysis and the experimental data can easily be expressed in terms of constraints and thus have prompted us to encode the problem within the constraint satisfaction problem paradigm.

3. Constraint Satisfaction Problem Algorithms

The constraint satisfaction problem consists of finding assignments of the variables $x_1, \ldots, x_n$ such that a set of constraints is satisfied. Each variable is restricted to a corresponding domain, i.e. $x_i \in D_i$. A *solution* to the problem is a particular assignment of variables that satisfies the constraints, and the result of a CSP algorithm is the list of all possible solutions.

The algorithm used here is based on the standard *backtracking* algorithm: "In this method, variables are instantiated sequentially. As soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available." [10]. The resulting computation is tree-like. Each branch of a node at level i ($i = 1$ at root) corresponds to an assignment of x_i that does not violate the constraints. Because the constraints prune the tree in an arbitrary way there is no guarantee that the tree is balanced. This becomes a concern when parallelizing the algorithm because it may cause poor load balancing.

In our application, there is one variable per nucleotide in the input sequence. The variable specifies the 3D position, orientation and conformation of the nucleotide. The conformation corresponds to the internal structure of the nucleotide, which can vary slightly depending on external factors. A problem with this formulation is that the domains are infinite. They can be made discrete and finite but they would have to be very large to attain a useful precision. The strategy we have adopted is to introduce problem-specific information to dynamically reduce the degrees of freedom of the domains [12]. The motivation is that secondary and tertiary interactions between nucleotides physically restrict their relative placement in space. For instance, the placement of a nucleotide restricts the placement of the next nucleotide in the sequence and any other nucleotide it interacts with. Thus, x_i's domain is dependent on the other variables.

It is convenient to express D_i as a function of the partial instantiation of the variables (i.e. the lower indexed variables $x_1, \ldots, x_{i-1}$). These *domain generating functions* (DGFs) can therefore only express backward dependencies. In our implementation, the DGF for D_i is a function that receives via its single argument the current assignment of $x_1, \ldots, x_{i-1}$ (as a list) and returns a list of the assignments to be considered for x_i. To help define the DGF of each nucleotide we have defined a few *parameterized* DGFs that capture the more common forms of nucleotide interactions. These parameterized DGFs are the functions: `reference`, `wc`, `wc-dumas`, `stacked3*`, `stacked5*`, `helix3*`, `helix5*`, and `P-O3*`. The purpose of `reference`, for example, is to place the first nucleotide at some arbitrary starting point, whereas `wc` is for a "Watson-Crick" type pairing of bases. The parameters of these functions are: the partial instantiation of variables, a label to name the nucleotide, the type of nucleotide (and its possible conformations) and the label of the other nucleotides involved in the interaction. See Appendix A.2 for a more detailed description of these functions and how the user prepares the input to the system. Figure A2 gives the definition of the `wc` parameterized DGF.

The core of the CSP algorithm is the `search` function shown in fig. 4 (for brevity we only give the Scheme encoding of the algorithm). This function is called for each node visited in the search tree. The argument `partial-inst` contains the

```
(define (search partial-inst domains constraint?)
  (if (null? domains)
    (make-singleton-queue partial-inst)
    (let ((remaining-domains (cdr domains)))

      (define (try-assignments lst)
        (if (null? lst)
          (make-empty-queue)
          (let ((var (car lst)))
            (if (constraint? var partial-inst)
              (let* ((subsols1
                       (search
                         (cons var partial-inst)
                         remaining-domains
                         constraint?))
                     (subsols2
                       (try-assignments (cdr lst))))
                (append-queues subsols1 subsols2))
              (try-assignments (cdr lst))))))

      (try-assignments ((car domains) partial-inst)))))
```

Figure 4. Scheme version of CSP algorithm.

assignment of variables up to that point (in effect, the path to the node from the root); `domains` is a list of the DGFs for the variables remaining to instantiate; and `constraint?` is the function for checking the constraints. For each node visited, `search` generates the domain for the current variable by calling the next DGF with the partial instantiation. Each possible assignment for the current variable is then explored by a recursive call to `search` and finally the resulting solution lists are concatenated. Queues are used to represent the list of solutions in order to have a constant time concatenation operation.

4. Program Development and Experiments

The project was carried out in three phases. First, the original Miranda implementation of the system was translated into Scheme. The system was then parallelized according to the Multilisp paradigm. Gambit [7] was chosen as the host Scheme implementation because it features an optimizing native-code compiler and it efficiently supports the Multilisp language. In the last phase, the Scheme version was translated to C in order to evaluate the costs of using Scheme.

The different versions of the program were tested on two realistic problems taken from a previous paper [13]. The first problem is the anticodon loop-stem structure from 1TRA. The second is a model proposed by Dumas *et al.* [4] for the pseudoknot structure. These two problems are relatively small when compared to other structures processed by MC-SYM (some structures take several days to solve on a high-performance workstation). The need for a fast system is emphasized by the interactive nature of the research process which typically requires that the same structure be processed repeatedly with slightly different parameters based on the results of previous runs.

Because we are primarily interested in the performance of the CSP algorithm, the programs simply count the number of solutions rather than sending them to a file. The list of solutions is nevertheless generated internally. Some vital statistics of these problems and the run time for the final sequential versions are given in fig. 5.

Problem	Nb. of Nucleo-tides	Nb. of Nodes Visited	Nb. of Pruned Branches	Nb. of Solu-tions	Run time in seconds		
					Miranda	Scheme	C
`anticodon`	17	1212	28621	179	17582	19.8	17.3
`pseudoknot`	23	5597	32900	50	23307	44.6	34.7

Figure 5. Results for sequential versions on the Apollo.

The timings in fig. 5 correspond to process time on an Apollo DN3500 (25 Mhz 68030 based) with 8 Mbytes of RAM running Domain/OS SR 10.3. The C version of the program was compiled with the native C compiler (`cc`) with optimizations enabled (`-O`). The Miranda execution was done with Miranda 2.015 [14] and a heap size of 3 Mbytes. The Scheme execution was done with Gambit 2.2 and a heap size of 3 Mbytes. This rather small heap size was the largest that avoided page faults on the Apollo. Gambit uses a simple stop-and-copy garbage collector based on Cheney's algorithm [3]. Consequently, each semispace is 1.5 Mbytes. The Multilisp runs reported in Section 7 were done with Gambit 2.2 on a BBN Butterfly GP1000 shared-memory multiprocessor [1] running Mach 1000 release 2.5.2. To avoid page faults, only 1 Mbytes of heap space was allocated per processor. Each of the GP1000's processors is a 16 Mhz 68020 with 4 Mbytes of local memory. Local memory is partitioned through software into private and shared sections. The program's code is copied to the private section of all processors and the heap of each processor is in the shared section (and is thus accessible from any processor). The cost for accessing a single word in remote memory is about 12 times larger than the cost for local memory. On the GP1000, a simple extension of the garbage collection algorithm makes it operate in parallel. As soon as some processor exhausts its free space, all processors are interrupted to start a garbage collection. Each processor traces its stack and copies into its own heap the objects it can reach. Race conditions are avoided by locking objects before they are copied. These locking operations increase the cost of garbage collection by a factor of roughly 1.5.

The final C, Scheme, and Multilisp versions of the program and the data sets for the anticodon and pseudoknot problems are available by anonymous FTP from `ftp.merl.com:/pub/LASC/nucleic.tar.Z`.

5. Translation to Scheme

The translation from Miranda to Scheme took 5 man-days and was done in two steps. The algorithm and overall structure of the Miranda program was preserved in the first step whereas in the second step we slightly modified the algorithm. Even though Miranda supports lazy-evaluation the program did not really need it, so when translating to Scheme all functions were assumed to be strict (i.e. no `delay` forms were introduced). The two programs are roughly the same size in number of lines of code. Initially we expected the Scheme version to be longer because of Miranda's terse syntax, but the use of macros in the Scheme code allowed substantial savings.

In addition, 5 man-days were spent optimizing the Scheme code. Originally the Scheme version was about twice as fast as the Miranda version. Gambit's profiler was helpful in fixing several sources of inefficiency:

- To our surprise, the program was spending a large proportion of its time in the bignum routines. This was traced to numerical type representation conversions performed by the generic arithmetic package. In particular, a relatively infrequent comparison between an inexact real (flonum) and an exact integer caused a conversion of the flonum argument to its exact representation (as a rational) in order to prevent the roundoff error that might have occurred had the exact integer been converted to a flonum. To avoid these costly conversions, numerical constants that might be involved in an inexact operation were rewritten as inexact reals. This reduced the run time by an unexpectedly large factor of 12.
- The run time was further reduced by 20% by inserting a declaration to remove type checks and open-code all simple primitives (e.g. `cons`, `vector-ref` but not `+`).
- Since generic arithmetic was no longer necessary, numerical type declarations were added to the program. For each numerical computation, the appropriate arithmetic operation was called (either flonum or fixnum specific). This decreased the run time by 33%.
- A few critical functions were rewritten as macros, further decreasing the run time by 5%.

At this point, the Scheme program was about 50 times faster than the Miranda version. Part of this difference can be attributed to the fact that the code generated by Miranda's compiler is interpreted. Another important factor is the overhead of lazy-evaluation. A closer examination of the Scheme program revealed three more ways of improving the program (these improvements were not carried over to the Miranda version):

- We noticed that the program could be reformulated slightly to expose some invariant computations on 3D transformation matrices, so the program was rewritten to use precomputed matrices. This decreased the run time by a factor of 8.

- At this point, roughly 19% of the time was spent in the garbage collector. When translating the program we were careful to ensure that as little garbage as possible would be generated. Garbage collections were still frequent due to the use of a functional programming style (which demands that functions allocate the result they return) and the extensive use of flonums (which Gambit implements with a 16 byte boxed representation containing a 64 bit floating-point number). To reduce the number of garbage collections we changed Gambit's representation of flonums to a more compact 8 byte representation. Garbage collection overhead went down to 11% of the total run time, which decreased by 11%. In principle, the garbage collection overhead can be lowered arbitrarily by increasing the heap size, but the limited amount of physical memory on the host computers precluded this option.

- Profiling the program showed that a substantial amount of time was spent in two numerical functions: the product of two 3D transformation matrices and the product of a 3D transformation matrix by a 3D vector. These were fairly short functions so we hand-coded them in assembler in such a way that the intermediate values were unboxed flonums. Garbage collection overhead went down to 8% of the total run time, which decreased by 38%.

The final version of the program runs roughly 500 and 900 times faster than the Miranda version for the pseudoknot and anticodon problems respectively.

6. Translation to C

The Scheme program was translated to C to measure the performance loss due to the choice of Scheme as the implementation language. The C version has the same structure but memory allocation is done differently. Instead of having functions dynamically allocate objects on the heap to return them to their caller, the space for the result is preallocated on the stack by the caller and a pointer passed to the function. The C version thus avoids heap allocation, garbage collection and the boxing of flonums.

The results from fig. 5 indicate that the Scheme version is a factor of about 1.15 to 1.3 slower than the C version. However, when the programs are run on the GP1000, the Scheme version running on a single processor is slower than the C version by a factor of 1.75 to 2. This larger difference can be accounted for by the smaller heap size which increases the garbage collection overhead to 40% of the run time. This shows that the performance of the garbage collector plays an important role in this application. It is reassuring that the Scheme code, with all the overheads mentioned above, combined with a small amount of assembler code, is within a factor of 2 of the performance attainable with an optimizing C compiler.

7. Translation to Multilisp

Translation to Multilisp was straightforward. It took a few minutes to obtain a parallel version of the program from the Scheme version. This program performed reasonably well for small number of processors but an additional 5 man-days were spent tuning the program for maximal performance.

Parallelization consisted of adding a single `future` form in `search` around the recursive call (line 12 in fig. 4) and a single call to `touch` around the reference to `subsols1` (line 18). Thus, with the exception of the root node, one task is created per node visited in the search tree. This placement of `future` and `touch` expresses parallelism between the exploration of each branch of the current node. The call to `touch` forces synchronization of the task exploring a branch with the task associated with the current node. Consequently, the parallel execution is of the "fork-join" variety, and no parallelism is exported outside of `search` (i.e. when `search` returns, all the tasks it has spawned have terminated).

This parallelization produces moderately coarse-grain tasks because of the relatively heavy computation required at each node. The average task size is 22 milliseconds for the anticodon problem and 10 milliseconds for the pseudoknot problem. Consequently the overhead of parallelization is small; in fact the run time of the Scheme and Multilisp versions is identical when run on a single processor with the same garbage collection algorithm. Gambit uses *lazy task creation* (LTC) to implement futures [6]. With LTC, `future` forms compile to a small number of machine instructions and it is only when another processor needs work that a larger price is paid to create and transfer a task. Thus, when the program is run on a single processor, no tasks are created and the overhead is almost zero. However, an optimized implementation of the more traditional *eager task creation* would probably give good results due to the moderate task granularity.

In [5] we found that, on the GP1000, contention for shared data can be a serious bottleneck when the number of processors is large. The GP1000 does not have coherent caches or combining circuitry in the memory interconnect so all accesses to a datum get serialized by the hardware. To reduce contention, Gambit automatically copies the program's code and constants to all processors. However, dynamically allocated data is placed on the processor performing the allocation so it becomes a bottleneck when it is accessed by simultaneously executing tasks. The program was modified in a few places to reduce contention. This did not change the run time on one processor but improved the run time for large number of processors. One modification was to rewrite the database of nucleotide conformations, which is heavily accessed, as a constant structure so that each processor would have a local copy. Another modification was in the way domains are described. Originally the parameterized DGFs were currified functions that returned DGFs. The DGFs were thus heap allocated closure objects which were a source of contention. The creation of these closure objects was avoided by lambda-lifting these functions by hand. Note however that some dynamically allocated shared structures still remain, namely, the list of DGFs, the partial instantiation, and the list of solutions.

Figure 6 gives the run times and speedup curves of the final program. The speedup curve for the anticodon problem is very good: slightly super-linear for up to 48 processors and then slightly below linear. The super-linear speedup can be explained by the decreasing garbage collection overhead as the number of processors increases. Since each processor has its own heap, the total heap size on n processors is n times larger than on one processor. However, Gambit maintains a few system data structures in the heap (e.g. symbol table and interpreter tables) so the free heap space on n processors is actually slightly more than n times that available on a single processor. Thus, garbage collections become slightly less frequent as the number of processors increases. For example, the proportion of the run time spent in the garbage collector drops from 49% on one processor to 40% when 2 processors are used and to 38% when 4 processors are used. The jumps in the speedup curve above 24 processors are due to the discrete nature of the garbage collector. At 32 processors and above, the program's run time is so short that the garbage collector no longer gets called.

The anticodon problem's degradation of performance for large number of processors is partly explained by the task spawning behavior at the beginning of the program's execution. Task spawning is directly dependent on the size of the domains and their ordering. The first task is spawned at .04 seconds and at .075 seconds only 32 tasks have been created. Thus there will be a significant amount of idle time at the beginning of the run, especially for any processor beyond the first 32. The effect of this idle time clearly becomes more important as the number of processors is increased (as explained by Amdahl's law). The ordering of the domains could be changed to spawn tasks sooner but this would have the detrimental effect of duplicating the work that is currently done once, at the start of the computation.

Speedup for the pseudoknot problem is not as great as for the anticodon problem. The speedup is roughly linear below 16 processors but at 24 processors it barely exceeds 16 and decreases slightly as more processors are used. At first we thought the less balanced search tree of the pseudoknot problem was causing an increase in task creation costs and idle time. However, profiling the program shows that these costs are fairly constant in the range of 16 to 64 processors. The real culprit is higher contention for the partial instantiation. Even though contention occurs in both problems it is more acute for the pseudoknot problem because the partial instantiation is accessed 5 times more frequently and, due to the domain ordering, the partial instantiation is mostly constructed on a single processor. Unfortunately, this contention problem is hard to solve because Multilisp does not provide constructs to control the placement of data and tasks.

The dynamic load balancing method used in Gambit performed well for both problems, even at high number of processors. At 64 processors, the idle time is on average 18% and 3.5% of total run time for the anticodon and pseudoknot problems respectively. The potential imbalance in the search tree is one of the prime motivations for adopting a programming system with fine grain dynamic load balancing. The static partitioning methods used in several other parallel programming systems

	anticodon Time (secs)	anticodon Speedup (T_1/T_n)	anticodon Nb. GCs	pseudoknot Time (secs)	pseudoknot Speedup (T_1/T_n)	pseudoknot Nb. GCs
C	25.000	2.11		49.400	2.39	
Scheme	44.245	1.19	28	97.703	1.21	64
Multilisp, n=1	52.810	1.00	28	118.024	1.00	64
2	22.967	2.30	13	54.318	2.17	31
4	10.822	4.88	6	27.647	4.27	15
8	5.480	9.64	3	14.874	7.93	8
16	2.681	19.70	1	8.778	13.45	5
24	2.174	24.29	1	7.156	16.49	4
32	1.154	45.76	0	7.204	16.38	3
40	1.149	45.96	0	7.325	16.11	2
48	1.035	51.02	0	7.409	15.93	2
56	.961	54.95	0	7.818	15.10	2
64	.902	58.55	0	8.074	14.62	2

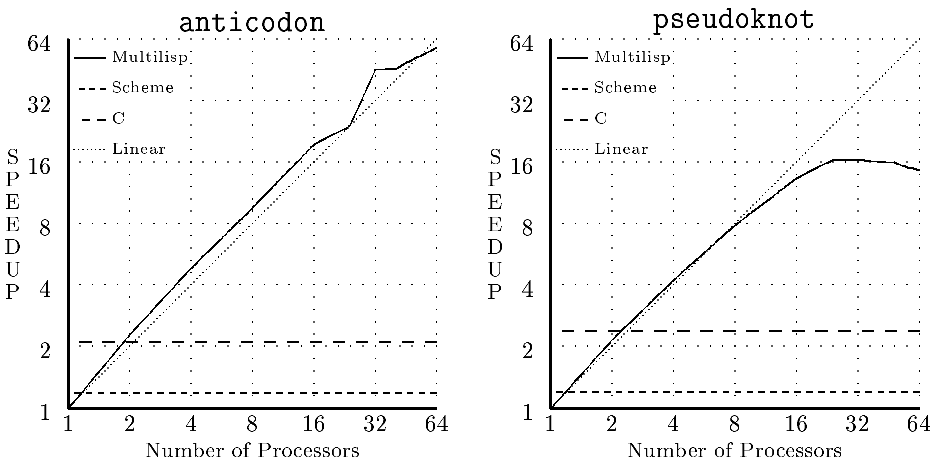


Figure 6. Timing results and speedup curves for anticodon and pseudoknot problems on the GP1000.

would lead to much more idle time when the search tree is not balanced, as is the case for the pseudoknot problem.

8. Conclusions

We have explored the parallelization of a functional program for determining the three-dimensional structure of nucleic acids. This is a "real-world" application that combines symbolic and numerical computation. The constraint satisfaction algorithm used in this application was relatively easy to parallelize using the Multilisp language. Nevertheless, it was necessary to tune the program in several ways to attain good performance on the GP1000 shared-memory multiprocessor. An important aspect was to identify and reduce the contention on shared data structures. In one case contention was high enough to limit the program's speedup to a factor of 16. Modifications in the original sequential algorithm, especially with regard to floating-point operations and garbage collection, were extremely important to achieve good absolute performance.

Our work demonstrates a practical symbolic application that can benefit from parallelism. On 64 processors, the Multilisp program is up to 27 times faster than a sequential version of the same program rewritten in C in an imperative style and compiled with an optimizing compiler.

Appendix

A.1. Comparative Sequence Analysis

There are positions of the secondary structure which are not conserved (or constant) among the nucleotide sequences of corresponding organisms. For such positions, we frequently observe strong covariation with the nucleotide position that is paired to it; the covariation is such that it preserves the complementarity of the bases. The diagram below shows the alignment of nucleotide sequences from different organisms of the anticodon stem-loop region of tRNAs. The lines join the two nucleotides of the same base-pair. The underlined region is the anticodon. We also observe such a pattern of covariation between nucleotides that are not located in a secondary structure element; those nucleotides may be involved in the tertiary interactions (e.g. the pair G19:C56 fig. 3 (b)).

	Anti–parallel base–pairings
Saccharomyces cerevisiae	...**CCAGA**CUGAAGA**UCUGG**...
Spiroplasma meliferum	**CCUGC**CUUGCAC**GCAGG**
Mycoplasma capricolum	**CCUCC**CUGUCAC**GGAGG**
Mycoplasma mycoides	**CACGG**UUUUCAU**CCGUG**
Spiroplasma meliferum	**UUUGA**UUGAAGC**UCAAA**
Streptomyces lividans	**ACGGC**CUGCAAA**GCCGU**
	30 35 40

A.2. Input Preparation: Anticodon Problem

This section explains how the user prepares the input to the system. As an example, we show the construction of a region of tRNA called the anticodon stem-loop that comprises nucleotides 27 to 43. This construction starts from three pieces of information. First, through comparative sequence analysis, nucleotides 27 to 31 are known to be base-paired to nucleotides 39 to 43 and form a double-helical domain. Second, nucleotides 34 to 38 are thought to be stacked on top of each other. Third, successive nucleotides in the sequence must be connected by $P - O3'$ covalent bonds.

The variable `anticodon-domains` is set to the list of DGFs (see fig. A1). The call `(reference rC 27 partial-inst)`, where `rC` is a rigid nucleotide conformation for the C ribonucleotide and 27 its label, will produce a list of one element, the single allowable placement of the first nucleotide (an arbitrary reference point). The call `(helix5* rC 28 27 partial-inst)` looks for nucleotide 27 (the first nucleotide) in `partial-inst` and calculates the 3D transformation matrix that puts C28 in space in such a way that it is connected to C27 and forms a regular **A** form helix. A similar computation is applied to nucleotides 29, 30, and 31. Then we break the linear extension and jump to the opposite strand and generate a Watson-Crick type base-pairing with the call `(wc rU 39 31 partial-inst)`. The nucleotides 40, 41, 42, and 43 are generated in the same manner as nucleotide 28. Up to now only the helical region has been accounted for and each DGF has returned a domain with one value; thus, there is only one possible partial instantiation. We now begin the computation for the loop region (nucleotides 32 to 38) for which there is some freedom in the placement of nucleotides. The call `(stacked3* rA 38 39 partial-inst)` generates two possible placements for the rigid nucleotide A38 to be stacked under nucleotide 39. Similarly, nucleotides 37, 36, 35, and 34 are generated, since no constraint has been involved the number of partial instantiations being explored is 32 (2^5). The DGFs introduced so far took one rigid nucleotide conformation and put it in one or two orientations in space; the `P-O3*` function takes a set of rigid conformations and tries three different placements for them (these rigid conformations represent intra-nucleotide variations such as torsion angles). There are 30 possible placements for nucleotide 32 and the same number for nucleotide 33. The number of leaves considered is thus 28,800 although the constraint (which requires the oxygen atom number $3'$ of nucleotide 33 to be no farther than 3 angströms from the phosphorus atom of nucleotide 34) causes only the 179 solutions to be visited.

Acknowledgments

We wish to thank Michigan State University and Argonne National Laboratory for the use of their Butterfly computer. This work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada and the Medical Research Council of Canada.

```
(define anticodon-domains
  (list
   (lambda (partial-inst) (reference rC  27    partial-inst))
   (lambda (partial-inst) (helix5*   rC  28 27 partial-inst))
   (lambda (partial-inst) (helix5*   rA  29 28 partial-inst))
   (lambda (partial-inst) (helix5*   rG  30 29 partial-inst))
   (lambda (partial-inst) (helix5*   rA  31 30 partial-inst))
   (lambda (partial-inst) (wc        rU  39 31 partial-inst))
   (lambda (partial-inst) (helix5*   rC  40 39 partial-inst))
   (lambda (partial-inst) (helix5*   rU  41 40 partial-inst))
   (lambda (partial-inst) (helix5*   rG  42 41 partial-inst))
   (lambda (partial-inst) (helix5*   rG  43 42 partial-inst))
   (lambda (partial-inst) (stacked3* rA  38 39 partial-inst))
   (lambda (partial-inst) (stacked3* rG  37 38 partial-inst))
   (lambda (partial-inst) (stacked3* rA  36 37 partial-inst))
   (lambda (partial-inst) (stacked3* rA  35 36 partial-inst))
   (lambda (partial-inst) (stacked3* rG  34 35 partial-inst));<-. Distance
   (lambda (partial-inst) (P-O3*     rCs 32 31 partial-inst));  | Constraint
   (lambda (partial-inst) (P-O3*     rUs 33 32 partial-inst));<-' 3.0 Angstroms
   ))

(define (anticodon-constraint? v partial-inst)
  (if (= (var-id v) 33)
    (let ((p   (atom-pos nuc-P (get-var 34 partial-inst))) ; P in nucleotide 34
          (o3* (atom-pos nuc-O3* v)))                      ; O3' in nucleotide 33
      (<= (pt-dist p o3*) 3.0))                            ; check distance
    #t))
```

Figure A1. Statement of the anticodon problem, including definition of the domains (and sequence) and the constraints. The call `(search '() anticodon-domains anticodon-constraint?)` produces the list of all solutions.

```
(define (wc nuc i j partial-inst) ; for Watson-Crick pairing of nucleotides i and j
  (let* ((ref (get-var j partial-inst))   ; find variable j
         (tfo (dgf-base wc-tfo ref nuc))) ; compute placement of nucleotide i
    (list (make-var i tfo nuc))))         ; create singleton domain

(define wc-tfo ; precomputed transformation matrix for Watson-Crick paired base
  '#(-1.0000  0.0028 -0.0019
      0.0028  0.3468 -0.9379
     -0.0019 -0.9379 -0.3468
     -0.0080  6.0730  8.7208))
```

Figure A2. The `wc` parameterized DGF.

References

[1] BBN Advanced Computers Inc., Cambridge, MA. *Inside the GP1000*, 1989.

[2] R. Bernstein, T. F. Koetzle, G. J. Williams, E. F. Meyer, M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi. The Protein Data Bank: A computer-based archival file for macromolecular structures. *Eur. Biochem.*, 80:319–324, 1977.

[3] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[4] P. Dumas, D. Moras, C. Florentz, and R. Giegé. 3-D Graphics Modelling of the tRNA-like 3'-end of Turnip Yellow Mosaic Virus RNA: Structural and Functional Implications. *J. Biomol. Str. Dynam.*, 4:707–728, 1987.

[5] M. Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University Department of Computer Science, 1993. Available as publication #869 from département d'informatique et recherche opérationnelle de l'Université de Montréal.

[6] M. Feeley. A message passing implementation of lazy task creation. In *Parallel Symbolic Computing: Languages, Systems, and Applications (US/Japan Workshop Proceedings).* Springer-Verlag Lecture Notes in Computer Science 748, November 1993.

[7] M. Feeley and J. S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming Languages and Computer Architecture*, Nice, France, June 1990.

[8] K. A. Frenkel. The human genome project and informatics. *Communications of the ACM*, 34(11):41–51, November 1991.

[9] D. Gautheret and R. Cedergren. Modeling the three-dimensional structure of RNA. *FASEB Journal*, 7:97–105, January 1993.

[10] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32–44, 1992.

[11] F. Leclerc, R. Cedergren, and A. D. Ellington. A three-dimensional model of the Rev Binding Element of HIV-1 derived from analyses of in vitro selected variants (submitted to Nature Structural Biology). *Nature Structural Biology.* This paper (in press) describes a protocol to apply energy minimization techniques to MC-SYM models.

[12] F. Major, G. Lapalme, and R. Cedergren. Domain Generating Functions for Solving Constraint Satisfaction Problems. *J. Funct. Prog.*, 1(2):213–227, 1991.

[13] F. Major, M. Turcotte, D. Gautheret, G. Lapalme, E. Fillion, and R. Cedergren. The Combination of Symbolic and Numerical Computation for Three-Dimensional Modeling of RNA. *Science*, 253:1255–1260, September 1991.

[14] Research Software Limited, Canterbury, England. *Miranda System Manual*, 1989.

[15] W. Saenger. *Principles of Nucleic Acid Structure.* Springer-Verlag, New-York, 1984.

[16] J. D. Watson, N. H. Hopkins, J. W. Roberts, J. Steitz Argetsinger, and A. M. Weiner. *Molecular Biology of the Gene,* Volume I & II. Benjamin Cummings, Menlo-Park, 1987.