Applications of the Aurora Parallel Prolog System to Computational Molecular Biology

Ewing Lusk
Argonne National Laboratory
Argonne, IL 60439
U. S. A.
lusk@mcs.anl.gov

Shyam Mudambi
ECRC GmbH
Arabellastr. 17
D-81925, Munich
Germany
mudambi@ecrc.de

Ross Overbeek
Argonne National Laboratory
Argonne, IL 60439
U. S. A.
overbeek@mcs.anl.gov

Péter Szeredi
IQSOFT
H-1142 Budapest
Teleki Blanka u. 15-17.
Hungary
szeredi@iqsoft.hu

Abstract

We describe an investigation into the use of the Aurora parallel Prolog system in two applications within the area of computational molecular biology. The computational requirements were large, due to the nature of the applications, and were carried out on a scalable parallel computer, the BBN "Butterfly" TC-2000. Results include both a demonstration that logic programming can be effective in the context of demanding applications on large-scale parallel machines, and some insights into parallel programming in Prolog.

1 Introduction

Aurora [8] is an OR-parallel implementation of full Prolog. The system is nearing maturity, and we are beginning to use it for application work. The purpose of this paper is to present the results of our experiences using it for computational molecular biology, an area in which logic programming offers a particularly appropriate technology.

The problems encountered in this area can be large in terms of data size and computationally intensive. Therefore one needs both an extremely robust programming environment and fast machines. Aurora can now provide the former. The fast machine used here is the BBN TC-2000, which provides fast individual processor speeds, a large shared memory, and a scalable architecture (which means that access to memory is non-uniform).

We begin with a brief discussion of why molecular biology is a particularly promising application area for logic programming. We then summarize some recent enhancements to Aurora as it has evolved from an experimental, research implementation to a complete, production-oriented system. We describe in some detail two different problems in molecular biology, and describe the approaches taken in adapting each of them for a parallel logic programming solution. In each case we present results that are quite encouraging in that they show substantial speedups on up to 42 processors, the maximum number available on our machine. Given the sizes of the problems that are of real interest to biologists, the speedups are sufficient to convert a batch-oriented research methodology into an interactive one.

2 Logic Programming and Biology

Many large-scale scientific applications running on parallel supercomputers are fundamentally numeric, are written in Fortran, and run best on machines optimized for operating on vectors of floating-point numbers. One notable exception is the relatively new science of genetic sequence analysis. New technologies for extracting sequence information from biological material have shifted the scientific bottleneck from data collection to data analysis. Biologists need tools that will help them interpret the data that is being provided by the laboratories. Logic programming, and Prolog in particular, is an ideal tool for aiding analysis of biological sequence data for several reasons.

- Prolog has built-in pattern expression, recognition, and manipulation capabilities unmatched in conventional languages.
- Prolog has built-in capabilities for backtracking search, critical in advanced pattern matching of the sort we describe here.
- Prolog provides a convenient language for constructing interactive user interfaces, necessary for building customized analysis tools for the

working biologist.

- (The Aurora hypothesis) Prolog provides a convenient mechanism for expressing parallelism.

Prolog has not traditionally provided supercomputer performance on scalable high-performance computers. The main point of this paper is that this gap is currently being closed.

3 Recent Enhancements to Aurora

Aurora has been evolving from a vehicle for research on scheduling algorithms into a solid environment for production work. It supports the *full* Prolog language, including all the normal intrinsics of SICStus Prolog, a full-featured system. Certain enhancements for advanced synchronization mechanisms, modifications to the top-level interpreter, and parallel I/O, have been described in [5]. Here we mention two recently-added features that were used in the present work.

3.1 Aurora on NUMA Machines

Aurora was developed on the Sequent Symmetry, which has a true shared-memory architecture. Such architectures provide a convenient programming model, but are inherently non-scalable. For that reason, the Symmetry is limited by its bus bandwidth to about 30 processors, and previously-published Aurora results were similarly limited. Recent work by Shyam Mudambi, continuing that reported in [9], has resulted in a port of Aurora to the BBN TC-2000, a current scalable architecture with Motorola 88000 processors. The results here were carried out on the machine at Argonne National Laboratory, where 42 processors at a time can be scheduled. We are currently planning to explore the performance of this version of the system on larger TC-2000's.

Aurora, with its shared-memory design, could be ported to the BBN in a straightforward way since the BBN does provide a shared-memory programming model. However, the memory-access times when data is not associated with the requesting processor are so much worse than when data accesses are local, it is critical to ensure a high degree of locality. This has been done in the Butterfly version of Aurora by a combination of duplicating read-only global data and allocating WAM stack space locally. Details can be found in [9]. The three-level memory hierarchy of the BBN also affects the interface to foreign subroutines, critical in the applications described here. In particular, it was necessary to change the original design of the C interface, which put dynamically-linked code in shared memory. Since on the Butterfly shared memory is not cached, we modified the design so that both code and data are allocated in non-shared memory and can therefore be cached.

3.2 Visualization of Parallel Logic

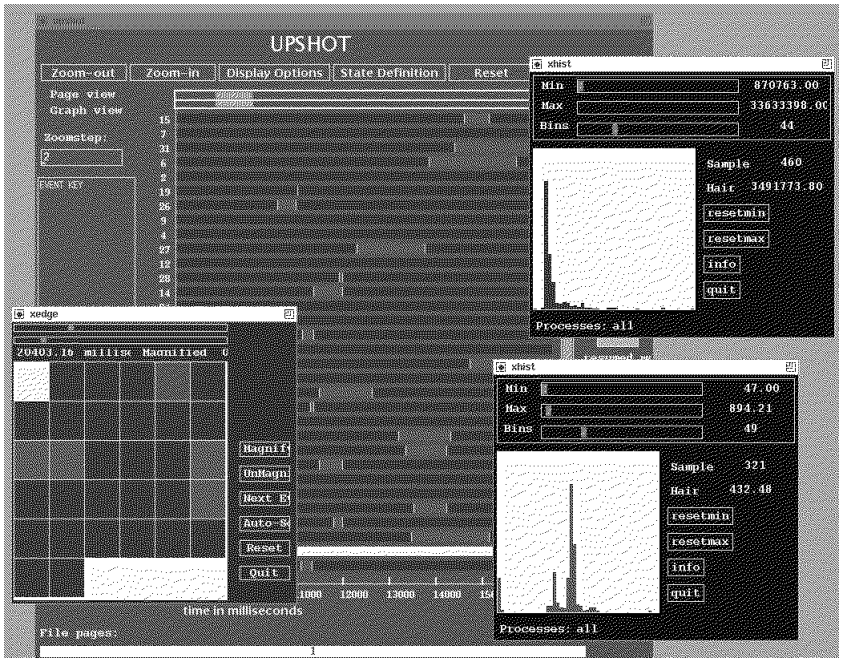

Figure 1: Investigating grain size with `upshot`

One can often predict the behavior of ordinary programs by understanding the algorithms employed, but the behavior of parallel programs is notoriously difficult to predict. Even more than sequential programs, parallel programs are subject to "performance bugs", in which the program computes the correct answer, but more slowly than anticipated. With this in mind, a number of tools have been added to Aurora in order to obtain a visual representation of the behavior of the parallel program. The first of these was `wamtrace`, which provided an animation of Aurora's execution. More recently, a number of other visualization systems have been integrated into Aurora. All of these tools provide post-mortem analysis of log files created during the run. Two of the tools have been used in tuning the applications presented here. They are `upshot`, which allows detailed analysis of events on a relatively small number of processes [6], and `gsx`, which is better suited to providing summary information on runs involving large numbers of processes. Other Aurora tools are `visandor` and `must`. An example snapshot of an `upshot` session is shown in Figure 1. Here we see a background window showing the details of individual processor activities, and, superimposed, subwindows providing a primitive animation of process states and histograms of state durations.

Output from `gsx` for one of the applications will be shown in Figure 3.

4 Use of Pattern Matching in Genetic Sequence Analysis

When trying to extract meaning from genetic sequences, one inevitably ends up looking for patterns. We have implemented two pattern matching programs—one for DNA sequences and one for protein sequences. Although these could certainly be unified, it is true that the types of patterns one searches for in DNA are quite distinct from those used for proteins. For a general introduction to genetic sequences, see [7].

4.1 Searching DNA for Pseudo-knots

DNA sequences are represented as strings composed from the characters {A,C,G,T}, each one of which represents a nucleotide. For example, an interesting piece of DNA might well be represented by `TCAGCCTATTCG...`. The types of patterns that are often sought involve the notion of *complementary substrings*, which are defined as follows:

1. The character complement of A is T, of C is G, of G is C, and of T is A.

2. The complement of a string $a_1a_2a_3\ldots a_n$ is $c_n\ldots c_3c_2c_1$, where c_i is the character complement of a_i.

To search for two 8-character substrings that are complementary and separated by from 3 to 8 characters, we would use a pattern of the form

```
p1=8...8  3...8  ~p1
```

which might be thought of as saying "Find a string of length 8 (from 8 to 8) and call it p_1, then skip from 3 to 8 characters, and then verify that the string that follows is the complement of p_1."

The significance of complementary substrings lies in the fact that complementary characters form bonds with each other, consecutive sets of which form biologically significant physical structures.

One particularly interesting type of pattern is called a *pseudo-knot*, which has the form

1. a string (call it p_1),

2. a filler,

3. a second substring (call it p_2),

4. a filler,

5. the complement of p_1,

6. a filler, and

7. the complement of p_2.

These patterns correspond to stretches of the DNA sequence that look like the diagram in Figure 2.

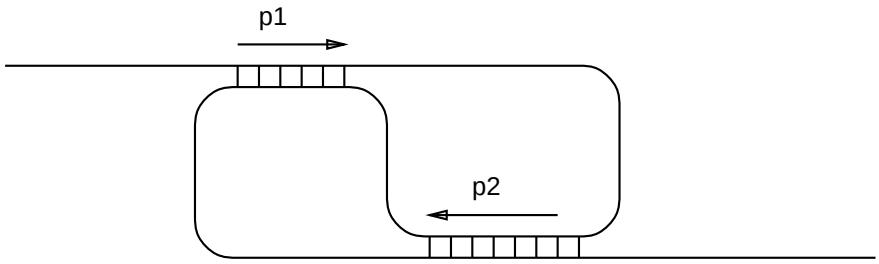

Figure 2: A pseudo-knot

Such patterns are often equally interesting when the complements of p_1 and p_2 are only approximate (i.e., most of the characters are complements, but there may be some that are not). We have implemented a language (based on the work of David Searls [10]) for expressing such patterns and for rapidly scanning DNA strings for matches. For a search to be well-specified, one has to express limits on the sizes of all substrings and fillers, as well as a level of tolerance when looking for complements. For example,

```
p1=9...9 2...9 p2=9...9 0...4 ~p1[1,0,0] 12...19 ~p2[1,0,0]
```

would represent a pattern in which p_1 and p_2 are 9 characters long, one character of each complement can mismatch (but there can be no insertions or deletions), and the three fillers are 2-9, 0-4, and 12-19 characters long, respectively.

4.2 Searching Protein Sequences

Protein sequences are strings over a 20-character alphabet, each character of which represents an amino acid. Amos Bairoch [1] has created a remarkable set of patterns that identify functionally significant sections of protein sequences. These patterns are composed of a sequence of pattern units, where a pattern unit can be

1. any character in a specified set, ([*list of characters*])
2. any character not in a specified set, ({*list of characters*})
3. a filler of length from a specified range. (`x`)

Pattern units can also have repetition counts (in parentheses). For example,

```
[LIVMFYWC]-[LIVM](3)-[DE](2)-x-[LIVM]-x(2)-[GC]-x-[STA]
```

means any of L,I,V,M,F,Y,W,C followed by three occurrences of any one of L,I,V,M, followed by two occurrences of any one of D,E, followed by any one character, etc.

This particular pattern identifies a "Purine/pyrimidine phosphoribosyl transferases signature". Given this somewhat more constrained matching problem, one can easily construct programs to search for such patterns. The most common problem is of the form "given a set of protein sequences and about 600 such patterns, find all occurrences of all patterns".

5 Evaluation of Experiments

5.1 The DNA Pseudo-knot Computation

A non-parallel program was already written (in Prolog) to attack the pseudo-knot problem. It ran on Sun workstations and had part of the searching algorithm written in C to speed up the low-level string traversal. The main contribution of Aurora in this application was to provide a Prolog interface (desirable since the user interface was already written in Prolog) to a high-performance parallel machine so that larger problems could be done.

The parallelization strategy was straightforward. A specific query asks for a restricted set of pseudo knots to be extracted from the database of DNA fragments. Almost all of the parallelism comes from processing the sequence fragments in parallel. Aurora detects and schedules this parallelism automatically.

The fact that we were porting a sequential program with C subroutines required us to take some care in handling the interface between Prolog and C. The structure of the mixed Prolog-C program is the following. The Prolog component parses a user query and through several interface routines passes the information on the pattern to be searched to the C-code. The Prolog side then invokes the actual search routine in C. The results of the search are transferred back to the Prolog component through appropriate interface routines again.

Several such searches are to be run in parallel independently of each other. For each search a separate memory area is needed in C. Since the original application wasn't programmed with parallel execution in mind, static C variables were used for storing the information to be communicated from one C subroutine to another. Conceptually we need the static C memory area to be replicated for each of the parallel processes.

In an earlier, similar application, we transformed the C program by replacing each static variable by an array of variables. Each of the parallel searches used one particular index position for storing its data. Indices were allocated and freed by the Prolog code at the beginning and at the end of the composite C computation tasks.

For the present application we chose a simpler route. Using sequential declarations we ensured that no parallel execution took place during a sin-

gle composite search task, i.e. it was always executed by a single process (worker). Consequently, all we had to ensure was that each process had a local piece of memory allocated for the C routines.

This goal was achieved in different ways on the two multiprocessor platforms we worked with. The Sequent Symmetry version of Aurora, like Quintus and SICStus, normally uses dynamic linking in the implementation of the foreign-language interface. Due to limitations in the Symmetry operating system, the dynamic linking process ignores the `shared` annotation on C variables, making them either all shared or all local. Because of this limitation Aurora loads the foreign code into shared memory on the Sequent, making all C variables shared by all the processes. Local allocation of variables can be thus achieved only by statically linking the C code with the emulator. This is what we did for the pseudo-knot application. The initial fork that creates multiple Aurora workers thus provided the required separate copies of the global variables.

On the BBN TC-2000, the situation was a little more complicated. In the first place, shared memory is not cached, so it was important to place the C code (and the Prolog WAM code as well) into the private memory of each processor. This was done, even with dynamic linking, by modifying Aurora so that after code was loaded, it was copied into each process's private memory. This provided both local copies of variables and cachability of all variables.

We ran a series of queries on the BBN Butterfly TC-2000, each of them designed to identify a collection of pseudo-knots (such as in Figure 2) of different sizes in a database of 457 DNA sequences, varying in length from 22 to 32329 characters. The following queries all ask for collections of pseudo-knots of varying sizes.

Table 1: Pseudo-knot queries

Goals	Patterns
ps1_2	p1=11...11 2...9 p2=11...11 0...4 p1[2,0,0] 14...21 p2[2,0,0]
ps2_1	p1=9...9 2...9 p2=9...9 0...4 p1[1,0,0] 12...19 p2[1,0,0]
ps2_2	p1=9...9 2...9 p2=9...9 0...4 p1[1,0,0] 12...19 p2
ps3	p1=7...7 2...9 p2=7...7 0...4 p1 10...17 p2
ps5	p1=11...11 2...20 p2=11...11 2...20 p1[2,0,0] 14...23 p2[2,0,0]

The specific pseudo-knot queries used in our tests are shown in Table 1. The times in seconds for these queries, run with varying numbers of processes on the TC-2000, are shown in Table 2. The figures in parentheses are speedups. Each value is the best of three runs.

The `gsx` summary of the one `ps3` query with 32 workers is shown in Figure 3. The various shades of gray (colors on a screen) indicate states of the Aurora parallel abstract machine. This particular picture indicates that

Table 2: Results of pseudo-knot query

Goals	Workers				
	1	16	32	36	42
ps1_2	2973.43	196.37(15.1)	104.46(28.5)	90.06(33.0)	79.43(37.4)
ps2_1	2775.75	185.10(15.0)	96.75(28.7)	86.76(32.0)	74.49(37.3)
ps2_2	2774.66	182.69(15.2)	96.66(28.7)	88.78(31.3)	73.45(37.8)
ps3	1771.56	120.62(14.7)	64.45(27.5)	59.51(29.8)	50.03(35.4)
ps5	16601.91	1047.12(15.9)	528.32(31.4)	472.28(35.2)	403.28(41.2)
Σ	26897.31	1733.68(15.5)	892.23(30.1)	799.56(33.6)	681.61(39.5)

the 32-worker machine was in the "work" state almost 90% of the time.

In these runs we used `upshot` to help us optimize the program. It showed us that the grain size of the parallel tasks varied enormously due to the variation in the size of the sequence fragments. If a large task is started late in the run, all other processes can become idle waiting for it to finish. We addressed this problem by pre-sorting the sequence fragments in decreasing order of length. This allows good load balancing from the beginning of the run to the end.

Table 3 shows the speed improvements obtained by pre-sorting sequence-fragments for selected queries. They are small but definitely significant.

Table 3: Percentage improvements due to sorting

Goals executed	Workers			
	16	32	36	42
`ps1_2`	2.6%	6.7%	12.0%	8.0%
`ps2_1`	3.1%	5.0%	6.7%	9.2%
`ps2_2`	3.2%	5.7%	8.3%	11.7%
`ps3`	3.3%	1.2%	0.1%	3.4%

5.2 The Protein Motif Search Problem

This problem involves finding all occurrences of some "protein motif" patterns in a set of proteins. The test data contains about 600 motifs and 1,000 proteins, varying in length from a few characters to over 3700. The algorithm for this task was designed with consideration for parallel execution; therefore we describe it in more detail.

The search algorithm is based on frequency analysis. The set of proteins to be searched is pre-processed to determine the frequency of occurrence of each of the characters representing the amino acids. Subsequently a proba-

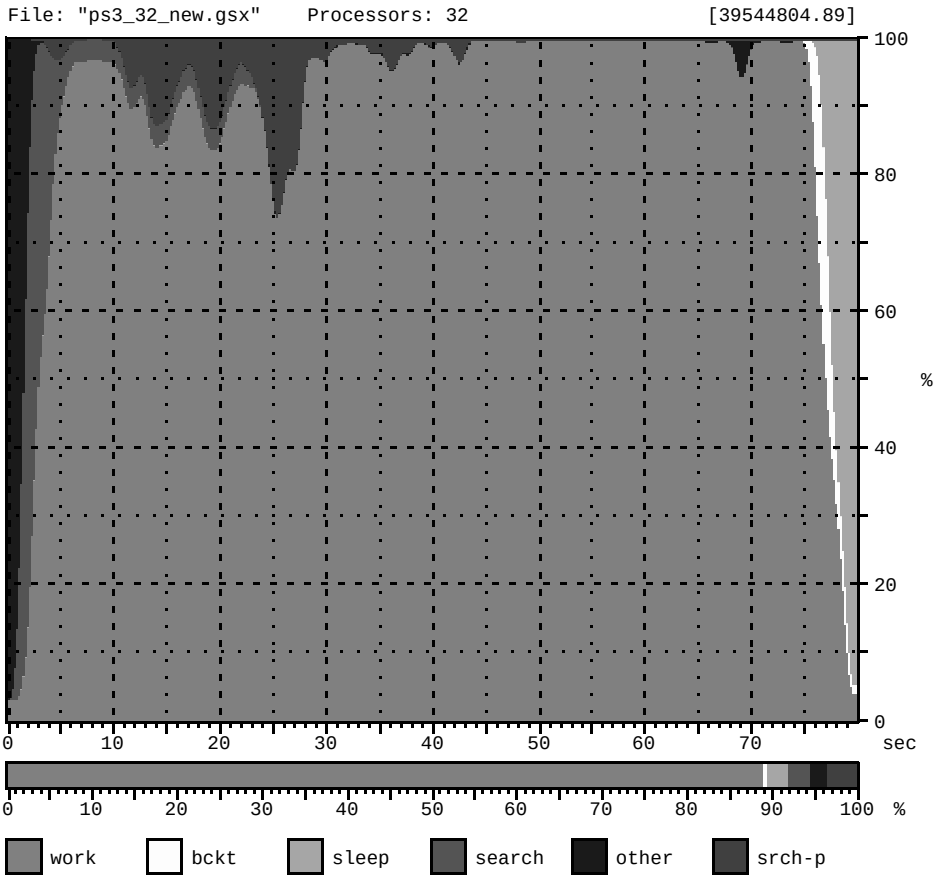

Figure 3: Efficiency of pseudo-knot query on TC-2000

bility is assigned to each character, inversely proportional to the frequency of occurrence in the proteins.

The protein motif patterns are originally given in the form described in Section 4.2. This form is then transformed to a Prolog clause, such as

```
prosite_pattern('PS00103',[any("LIVMFYWC"),any("LIVM",3),
                    any("DE",2),arb,any("LIVM"),arb(2),
                    any("GC"),arb,any("STA")]).
```

For each pattern a "most characteristic" character position is selected, i.e. the position with the smallest probability of matching. As the pattern matching algorithm will start at this most characteristic position, the pattern is split to three parts: the characters before, at, and after the given position, with the "before" part reversed. The split form is represented by one or more Prolog clauses, one for each choice for the characteristic position. The ASCII code of the character in this position is placed as the first argument of the clause, for the purpose of fast indexing.

The above example pattern has the `any("GC")` position as the most

characteristic, and thus the following representation is produced (note that 67 is ascii C and 71 is ascii G):

```
/*prosite_dpat(Code, BeforeReversed, After, Name).*/

prosite_dpat(67,[arb(2),any("LIVM"),arb,any("DE",2),
                    any("LIVM",3),any("LIVMFYWC")],
                [arb,any("STA")],'PS00103').
prosite_dpat(71,[arb(2),any("LIVM"),arb,any("DE",2),
                    any("LIVM",3),any("LIVMFYWC")],
                [arb,any("STA")],'PS00103').
```

The search algorithm has three levels of major choice-points: selection of a protein, selection of a pattern and the selection of a position within the protein where the matching is attempted. The introduction of the characteristic position helps in reducing the search space by efficient selection of patterns that can be matched against a given position in a given protein. This implies a natural order of search shown in Figure 4.

Select Proteins: select a protein, say P,

Select Positions: select a position within this protein, say N, with a character C,

Select Patterns: select a pattern M which has C as the most characteristic element,

Check: check if pattern M matches protein P before and after position N.

Figure 4: The search space of the protein motif problem

We have implemented the protein motif search program based on the above algorithm and data representation. With the test case containing 1000 proteins and 600 patterns, in principle there is abundant parallelism in exploring the search space. Over 11000 matches are found in the database, so collecting the solutions also requires some caution.

Let us first examine how easy it is to exploit the parallelism found in the problem. Table 4 shows the execution times in seconds (and the speedups in parentheses) for the search program run in a failure driven loop. This means that the the search space is fully explored, but solutions are not collected. The first line of the table is for the original database of proteins. Although the results are very good for smaller numbers of workers, the speedup for 42 workers goes below 90% of the ideal linear speedup.

Now we examine the search space as shown in Figure 4, and try to pinpoint the reasons for the disappointing speedups. First, we can deduce that the coarse grain parallelism of the **Proteins** level is not enough to

Table 4: Exploring the Search Space in the Protein Motifs Query

Program variant	Workers 1	16	24	36	42
1.	2962.50	186.76(15.9)	127.14(23.3)	89.50(33.1)	79.72(37.2)
2.	2965.79	185.49(16.0)	123.83(24.0)	82.89(35.8)	71.03(41.8)
3.	2952.18	185.39(15.9)	125.10(23.6)	86.31(34.2)	75.67(39.0)
4.	2952.54	184.45(16.0)	123.00(24.0)	82.22(35.9)	70.80(41.7)
1 = original program 2 = sorted data 3 = bottom-up 4 = both					

allow for uniform utilization of all workers throughout the computation. Second, the finer level parallelism at the level of **Positions** is not exploited sufficiently to compensate for the uneven structure of work on the protein level.

The first deficiency of our program can be easily explained by reasons similar to those already described for the pseudo-knot computation: the different size proteins represent different amounts of work. Consequently, if a larger protein is processed towards the end of the computation, the system may run out of other proteins to process in parallel before the longer computation is finished. The solution of sorting the proteins in decreasing order of size has been applied and the results are shown in second row of Table 4. The speedups are almost linear, being less than 1% below the ideal speedup. Although this change is enough on its own to solve our problem of poor speedup, the issue of exploiting finer grain parallelism on the level **Positions** is also worth exploring.

The second level of choice in our search problem is the selection of a position within the given protein to be used as a candidate for matching against the most characteristic amino acid in each pattern. Since proteins are represented by lists of characters standing for amino acids, this choice is implemented by a recursive Prolog predicate scanning the list. The following is a simplified form of this predicate[1], similar to the usual `member/2`.

```
find_match([C|Right],Left) :-
        find_a_matching_pattern(C,Right,Left).
find_match([C|Right],Left) :-
        find_match(Right,[C|Left]).
```

The or-parallel search space created by this program is shown in part (a) of Figure 5. When a search tree of this shape is explored in a parallel system such as Aurora, a worker entering the tree will first process the leftmost leaf and will make the second choice at the top choice-point available to other

[1] We just show the search aspects of the program, omitting those parts which deal with returning the solution.

workers. This choice is either taken up by someone else, or the first worker takes it after finishing the leftmost leaf. In any case it can be seen that the granularity of tasks is rather fine (the work at a leaf is often only a few procedure calls long) and the workers will be getting in each other's way, causing considerable synchronization overheads.

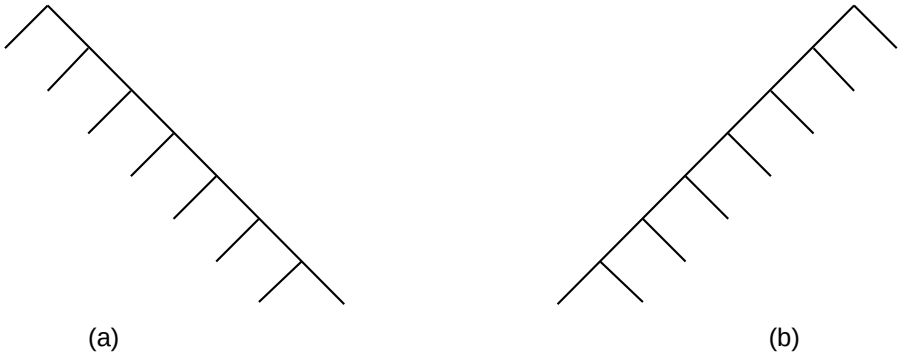

Figure 5: Alternative parallel search trees

The exploration of the search space is thus done by descending from top to bottom, an approach analogous to the topmost scheduling strategy of early parallel Prolog implementations [3, 4]. Later research showed that this type of scheduling is rather inefficient for finer grain granularity problems [2]. In our problem, topmost scheduling is forced on the system by the shape of the search tree, irrespectively of the scheduling strategy of the underlying system[2].

To avoid top-to-bottom exploration, one would like the system to descend first on the recursive branch, thus opening up lots of choice-points. The left-to-right exploration of alternatives within a choice-point is, however, an inherent feature of most or-parallel systems. In fact, rather than to require the system to change the order of exploration, it is much easier to change the order of clauses in the program and thus direct the system to explore the search tree in a different manner. This is analogous to the way users of a sequential Prolog system influence the search by writing the clauses of a predicate in appropriate order.

In the case of the present search problem the desired effect of bottom-to-top exploration can be achieved by transposing `find_match`'s two clauses. This way the search tree will have the shape shown in part (b) of Figure 5. The first worker entering the tree will run down the leftmost branch opening up all choice-points, which then will be processed by backtracking from bottom towards the top. Several workers may cooperate in this process, thus sharing the work through public backtracking rather then the more expensive means of "major task switching" as described in [2].

The third row of Table 4 shows the effect of bottom-to-top exploration

[2]We actually used the Bristol scheduler with bottommost scheduling.

technique applied to the original database of proteins. The results are somewhat better than in the corresponding first row, although not as good as the ones achieved by tuning the coarse grain parallelism (second row). The final row of the table shows the results obtained with both improvements applied, showing a small improvement over the second row.

Having explored the issues of parallel search space traversal, let us now turn to the problem of collecting the solutions. We have experimented with several variants of this program; the test results are shown in Table 5. The very first row, showing the times for the failure driven loop, is identical to the last row of Table 4 and is included to help assess the overheads of collecting the solutions. The runs shown here were done using the bottom-to-top variant of the program with a sorted database. The last row of the table is our final, best variant, showing a speedup of 40.3 with 42 workers, an efficiency of 96%.

Table 5: Results of the Protein Motifs Query

Goals executed	Workers 1	16	24	36	42
fail loop	2952.54	184.45(16.0)	123.00(24.0)	82.22(35.9)	70.80(41.7)
1 setof	3072.26	232.26(13.2)	180.64(17.0)	150.16(20.5)	148.08(20.7)
2 setof's	3114.44	198.96(15.7)	136.61(22.8)	97.39(32.0)	86.77(35.9)
2 findall's	2971.16	188.30(15.8)	126.38(23.5)	86.26(34.4)	73.81(40.3)

In the first variant of the program we used a single `setof` predicate to collect all the solutions (see the `single setof` row in Table 5). Our visualization and tuning tools (Figure 1) showed us that the low efficiency we initially attained was mainly due to the large number of solutions generated by the query. Since the parallel version of the `setof` operation collects solutions serially at the end of the query, this led to a long "tail" at the end of the computation in which one worker was collecting all the solutions. In order to parallelize the collection of solutions we replaced the single `setof` by a nested pair of `setof`'s, i.e. the solutions for each protein were collected separately, and a list of these solution-lists was returned at the top level of the search. The data for this improved version is shown in the `double setof` row of Table 5. Though this change resulted in a slight increase in the sequential time, the overall parallel times improved a great deal since more of the solution-gathering activity could proceed in parallel.

In this second variant of the program a separate `setof` predicate is invoked for each protein. Since the `setof` scans its arguments in search for free variables, this involves scanning the huge list representing the protein. To avoid this overhead, the `setof` calls were replaced by calls to `findall`, resulting in further improvement in performance, in terms of absolute time and speedup as well. This final result is shown in the `double findall` row

of Table 5).

6 Conclusion

The success of parallel logic programming requires three things: scalable parallel machines (bus-based "true" shared-memory machines are being eclipsed by fast uniprocessor workstations), a robust parallel logic programming system, and appropriate applications. Our preliminary experiments here indicate that the BBN TC-2000, the Aurora parallel Prolog system, and two applications in molecular biology represent such a combination.

Beyond the potential contribution of parallel logic programming to large-scale scientific applications, the work reported on here is interesting because it reflects a new and different phase of the Aurora parallel Prolog project. In earlier papers on Aurora, we (and others) have written about Aurora's design goals, its system architecture, and alternative scheduling algorithms. Each of these was an interesting and fruitful research topic in its own right. This paper reports on the *use* of Aurora. During this work there was no tinkering with the system (except for the machine-dependent memory management work described in Section 5.1) or comparison of alternative scheduling mechanisms.

The original goal of the Aurora project was to determine whether Prolog by itself could be an effective language for programming parallel machines. C and Fortran programmers still must concern themselves with the explicit expression of a parallel algorithm, despite considerable efforts to produce "automatic" parallelizing compilers. It was hoped that the parallelism *implicit* in Prolog could be exploited by the compiler and emulator more effectively than is the case with lower-level languages. Our experiments here by and large confirm this hypothesis: in both the pseudo-knot and the protein motif problems, good speedups were obtained with our initial Prolog programs, written as if for a sequential Prolog system. On the other hand, we also found that performance could be improved by altering the Prolog code so as to "expose" more of the parallelism to the system (top-to-bottom vs. bottom-to-top scanning), eliminate unnecessary sequential bottlenecks (two `setof`s vs. one) and permit load-balancing (pre-sorting of sequences). That we were able to do this fine-tuning at the Prolog level is a measure of success of the research into scheduling policies: when we gave the current Aurora system more freedom, it was able to exploit it to increase the amount of parallel execution.

Thus Aurora remains a tool for parallel algorithm research, but at the Prolog level as opposed to the C level. At the same time, its ability to convert hours of computation into minutes of computation on scientific problems of real interest attests to its readiness for a production environment.

Acknowledgments

Ewing Lusk and Ross Overbeek were supported in part by the Office of Scientific Computing, U.S. Department of Energy, under contract W-31-109-Eng-38. Shyam Mudambi's work was done while the author was at Knox College, Galesburg, Illinois. Péter Szeredi and Ewing Lusk were both partially supported by the U.S.-Hungarian Science and Technology Joint Fund under project No. 031/90.

References

[1] A. Bairoch. PROSITE: a dictionary of sites and patterns in proteins. Nucleic Acids Res. 19:2241-2245(1991).

[2] Anthony Beaumont, S Muthu Raman, Péter Szeredi, and David H D Warren. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science, Vol 506.

[3] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605, MIT Press, August 1988.

[4] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435, MIT Press, June 1989.

[5] Mats Carlsson, Ewing L. Lusk, and Péter Szeredi. Smoothing rough edges in Aurora (Extended Abstract). In *Proceedings of the First COMPULOG-NOE Area Meeting on Parallelism and Implementation Technology*. Technical University of Madrid, May 1993.

[6] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with `Upshot`. Technical Report ANL–91/15. Argonne National Laboratory, 1991.

[7] Wen-Hsiung Li and Dan Graur. Fundamentals of Molecular Evolution. Sinauer and Associates, 1991.

[8] Ewing Lusk, Ralph Butler, Terence Disz, Robert Olson, Ross Overbeek, Rick Stevens, D.H.D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumił Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[9] Shyam Mudambi. Performance of Aurora on NUMA machines. In Koichi Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference on Logic Programming*, pages 793–806, MIT Press, 1991.

[10] David Searls. Investigating the Linguistics of DNA with Definite Clause Grammars. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming: Proceedings of the 1989 North American Conference*, pages 189–208, MIT Press, 1989.