Contributions To Or-Parallel Logic Programming

PhD Thesis

Péter Szeredi

Technical University of Budapest

December 1997

Contents

Abstract

This thesis describes work on Aurora, an or-parallel logic programming system on shared memory multiprocessors. The Aurora system, supporting the full Prolog language, was developed in an international collaboration, called the Gigalips project.

The contributions described in the thesis address the problems of implementation, language and applications of or-parallel logic programming.

The Aurora *implementation* contains two basic components: the engine, which executes the Prolog code; and the scheduler, which organises the parallel exploration of the Prolog search tree. As our first investigation in this area, we carried out a detailed performance analysis of Aurora with the so called Manchester scheduler. Using the results of this study, we designed the Bristol scheduler, which provides a flexible scheduling algorithm and improved performance on programs involving pruning. We also defined a strict engine-scheduler interface, which reflects the main functions involved in or-parallel Prolog execution. The interface has been used in all subsequent Aurora extensions, as well as in the Andorra-I system.

We have studied the problems of Prolog *language extensions* related to parallel execution. We have experimented with parallelisation of programs relying on non-declarative Prolog features, such as dynamic predicates. We have designed and evaluated higher level language constructs for the synchronisation of parallel execution. We have also designed a parallel algorithm for solving optimisation problems, which supports both the minimax algorithm with alpha-beta pruning and the branch-and-bound technique. We have proposed language extensions to encapsulate this general algorithm.

We have worked on several *applications* of Aurora. Two large search problems in the area of computational molecular biology were investigated: search of pseudo-knots in DNA sequences and search of protein sequences for functionally significant sections. A large medical thesaurus was also transformed into Prolog, and evaluated on Aurora. Finally a scheme of a single WWW server capable of supporting multiple concurrent Prolog searches was developed using Aurora.

The work of the author described in this thesis had a significant impact on the Aurora implementation. It has also demonstrated that the system can be further extended to address special problem areas, such as optimisation search. The applications explored have proven that an or-parallel Prolog system can produce significant speedups in real-life applications, thus reducing hours of computation to a few minutes.

Acknowledgements

I was introduced to the topic of parallel logic programming by David H. D. Warren, when I joined his research group at the University of Manchester in 1987, which a year later moved to the University of Bristol. I am indebted to David, for introducing me to this topic, for constant encouragement and help, even after my leaving England. I enjoyed working with all my colleagues at Manchester and Bristol. I would like to specially thank Tony Beaumont, Alan Calderwood, Feliks Kluźniak, and Rong Yang for numerous discussions and help with the work described in this thesis.

When I joined David's group, I was fortunate to be immediately drawn into an informal collaboration, called the Gigalips project, which involved the Argonne National Laboratory (ANL), USA and the Swedish Institute of Computer Science (SICS). I learned how to carry discussions through electronic mail and how to run and debug programs at remote sites. I enjoyed very much the hacking sessions, when the contributions developed at the distant sites were merged and started to work together. Again, I would like to thank all my Gigalips colleagues, but especially Mats Carlsson and Ewing Lusk who became personal friends. I am very sad that my thanks to Andrzej Ciepielewski cannot reach him any more.

On my return to Hungary in 1990, I joined IQSOFT Ltd, led by Bálint Dömölki. I am indebted to Bálint and the management of IQSOFT, for the support they gave to this continued research. I would like to thank my colleagues at IQSOFT for helping me in this work, especially Zsuzsa Farkas, Kati Molnár, Rob Scott and Gábor Umann.

Work described in this thesis was supported by grants from the UK Science and Engineering Council, the European Union Esprit and Copernicus programmes, the US-Hungarian Science and Technology Joint Fund, and the Hungarian National Committee for Technical Development.

Chapter 1

Introduction

This thesis describes work in the area of or-parallel logic programming, carried out during years 1987–1996.

This chapter gives an overview of the thesis. First, the basic ideas of logic programming and its parallel implementations are outlined. Next, a summary of the thesis is presented, showing the problems to be solved, the approach to their solution, the results achieved, and their utilisation. Finally, the structure of the remaining part of the thesis is outlined.

1.1 Preliminaries

This section gives a brief overview of the problem area of the thesis: parallel logic programming. We first introduce the two areas involved: parallel computing and logic programming. We then discuss approaches to parallel execution of logic programs and their implementations. We conclude this section with an overview of the Aurora or-parallel Prolog system which is the subject of the thesis.

1.1.1 Parallel programming

It is a well known fact that the size of software systems grows very rapidly. Larger software requires bigger and bigger hardware resources. However, the speed of current hardware is approaching absolute physical limits. We are reaching a phase when further increase in speed can only be gained by parallelisation.

Parallelism in computations can be exploited on various levels. For example, there can be parallelisation within a single processor; one can have a computer with multiple processors working in parallel; or one can use computer networks distributed worldwide, as parallel computing resources.

Multiprocessor systems are positioned in the middle of this wide range. These are computers with multiple CPUs, coupled either tightly (e.g. through a shared memory), or loosely (e.g. using message passing). In the last few years, multiprocessor systems have became more widespread; recently even personal computer manufacturers have started to offer shared memory multiprocessor PCs.

The simplest way to make use of multiprocessor systems is to have the processors perform independent tasks in parallel (through e.g. multitasking operating systems). But what can we do, if we want to use the available computing resources to perform a single huge task as fast as possible? In this case, we have to parallelise the algorithm for the task; i.e. we have to break it down into several smaller co-operating parts.

There are two basic ways of parallelising an algorithm. This can either be done *explicitly* or *implicitly*. In the first case, the programmer has to decide which parts of the algorithm are to be executed in parallel, and how should they communicate with each other. Although tools and techniques have been developed to help produce parallel programs, writing such algorithms still proves to be very difficult. This is because the programmer has to understand and control the workings of several communicating instruction threads. Moreover, the debugging of parallel programs is very difficult, as the runs are highly time-dependent: two executions of the program will almost definitely result in different timing, and thus in different communication patterns.

In the second case of *implicit* parallelism, automatic transformation or compilation tools perform the selection

of tasks to be done in parallel, and organise their communication. The programmer does not need to worry about parallelism, he or she can write the algorithm as if it was to be executed on a single processor. The automatic parallelisation tools transform the algorithm to an equivalent parallel program.

For traditional, *imperative* programming languages automatic parallelisation is a very difficult task. This is because at the core of such languages is the variable assignment instruction, and programs are essentially *sequences* of such assignments. That is why automatic parallelisation tools for imperative languages are normally restricted to some special constructs, such as for-loops.

As opposed to imperative languages, *declarative* programming languages use the notion of a mathematical variable: a single, possibly yet unknown value. This is often referred to as the "single assignment principle". Declarative languages are thus much more amenable to automatic exploitation of parallelism, while, of course, still leaving room for explicit parallelisation, as in [38]. Implicit parallelism is especially important for *logic programming*, a programming paradigm building on mathematical logic.

1.1.2 Logic programming

Logic programming was introduced in early 1970's by Robert Kowalski [30], building on resolution theorem proving by Alan Robinson [34]. The first implementation of logic programming, the *Prolog* programming language, was developed by the group of Alain Colmerauer [37].

The basic principle of logic programming is that a program is composed of statements of predicate logic, restricted to the so called Horn clause form. A simple Prolog program below defines the `grandparent` predicate using the notion of `parent`.

```
grandparent(GrandChild, GrandParent) :-
            parent(GrandChild, Parent),
            parent(Parent, GrandParent).
```

Here the `:-` connective should be read as implication ($\leftarrow$), and the comma as conjunction. Capitalised identifiers stand for variables, lower case identifiers denote constants, function or predicate names. The above statement can be read as the following:

`GrandChild`'s grandparent is `GrandParent` **if**
(there exists a `Parent` such that)
`GrandChild`'s parent is `Parent`, **and**
`Parent`'s parent is `GrandParent`.

This is the declarative reading of the program. But the same program also has a procedural meaning:

To prove the statement `grandparent(GrandChild, GrandParent)`
prove the statements:
`parent(GrandChild, Parent)` and
`parent(Parent, GrandParent)`.

In such a procedural interpretation, statements to be proven are often referred to as *goals*.

Note that the order of proving the two statements in the grandparent procedure is not fixed, (although one execution order can be more efficient than another).

Let us now look at the definition of `parent`hood, which uses a disjunction (denoted by a semicolon).

```
parent(Child, Parent) :-
            (   mother(Child, Parent)
            ;   father(Child, Parent)
            ).
```

This statement can be read declaratively as:

`Child`'s parent is `Parent` **if**
its mother is `Parent`
or its father is `Parent`.

The procedural reading states that, to prove a parenthood statement, one has to prove either a motherhood or a fatherhood statement. Such a situation, when one of several possible alternatives can be executed, is called a *choice point*. One can visualise a choice point as a node of a tree with branches corresponding to alternatives. A set of nested choice points constitute the search tree, which the execution has to explore in order to solve a problem.

The program for parenthood can also be written as:

```
parent(Child, Parent) :-
           mother(Child, Parent).
parent(Child, Parent) :-
           father(Child, Parent).
```

Here we have two alternative clauses, both of which can be used to prove a parenthood relation. It is thus natural to define a procedure as the set of clauses for the same predicate, which specify how to reduce the goal of proving a statement to conjunctions and disjunctions of other such goals.

Although the procedural reading of logic programs does not fix the order of execution, most logic programming languages do prescribe an order. In Prolog both the **and** and **or** connectives are executed strictly left-to-right. Correspondingly, Prolog traverses the search tree in depth-first, left-to-right order. The fact that the programmer knows exactly how the proof procedure works, makes this approach a programming, rather than a theorem proving, discipline.

While the core of Prolog is purely declarative, it is important to note that the language has several impure, non-declarative features. Perhaps the most important is the *cut* operation, denoted by an exclamation mark (`!`), which prunes certain branches of the search tree. Other non-declarative elements include built-in predicates for input-output and for program modification. An example of the latter is the built-in `assert`, with which a new clause can be added to the program, during execution. For example the goal `assert(mother(abel, eva))` extends the program with the clause `mother(abel, eva)`. Modifiable predicates, such as `mother` in this example, are called *dynamic predicates*.

1.1.3 Parallel execution of logic programs

As said earlier, parallel execution of a program requires that the task to be performed is split into subtasks that can be executed on different processors. For logic programs, such a decomposition is very natural: a goal is decomposed into some other goals built with connectives **and** and **or**. Correspondingly there are two basic kinds of parallelism in logic programming: *and-parallelism* and *or-parallelism*.

One can distinguish between *independent* and *dependent* and-parallelism. The former occurs if two subgoals of a clause do not share any variables. For example, the goal of matrix-vector multiplication can be decomposed into two independent subgoals: computing the scalar product of the first row of the matrix and the vector; and computing, recursively, the matrix-vector product of the remainder of the matrix and the vector.

We speak about dependent and-parallelism in a clause, if two subgoals share a variable. For example, in the `grandparent` example, the two `parent` subgoals share the `Parent` variable. The two goals can thus be started in parallel, but as soon as one of them instantiates the common variable, the other has to be notified about this. The goal which instantiates the variable can be thought of as the producer and the other as the consumer of the variable. In more complex cases the producer-consumer interaction can be used for implementing a communication stream between the subgoals. This form of parallelism is also called *stream-parallelism*.

To exploit *or-parallelism*, one can use multiple processors to explore alternative branches of the search tree. For example, when executing the goal `parent(abel, Parent)`, one of the processors can attempt to solve the goal `mother(abel, Parent)`, and the other the goal `father(abel, Parent)`. It is inherent in or-parallelism, that the two subtasks can be solved independently.

We now discuss the case of or-parallelism in more detail, as it forms the basis of this thesis. Let us look at a slightly more complicated example. The task is to choose a holiday destination reachable from Budapest by a single flight, or at most two connecting flights. We have a database of flights in the form of Prolog clauses:

```
flight(budapest, venice, ...).
```

```
flight(budapest, paris, ...).
flight(paris, nice, ...).
flight(paris, london, ...).
...
```

These clauses are so called unit clauses, which have no preconditions, and so the `:-` connective is omitted. The third argument of the `flight` predicate contains further timetable details of the flight (such as departure and arrival time, days of operation, etc.).

The following is an outline of a program for finding appropriate holiday destinations:

```
destination(City):-
          flight(budapest, City, TTData),
          appropriate(City, [TTData]).
destination(City):-
          flight(budapest, Transfer, TTData1),
          flight(Transfer, City, TTData2),
          appropriate(City, [TTData1,TTData2]).
```

Here the `appropriate` predicate has the destination `City` as its first, and the list of timetable data as its second argument. It holds, if the given selection of flights satisfies some further unspecified criteria.

The search tree of the above program is depicted in Figure 1.1.

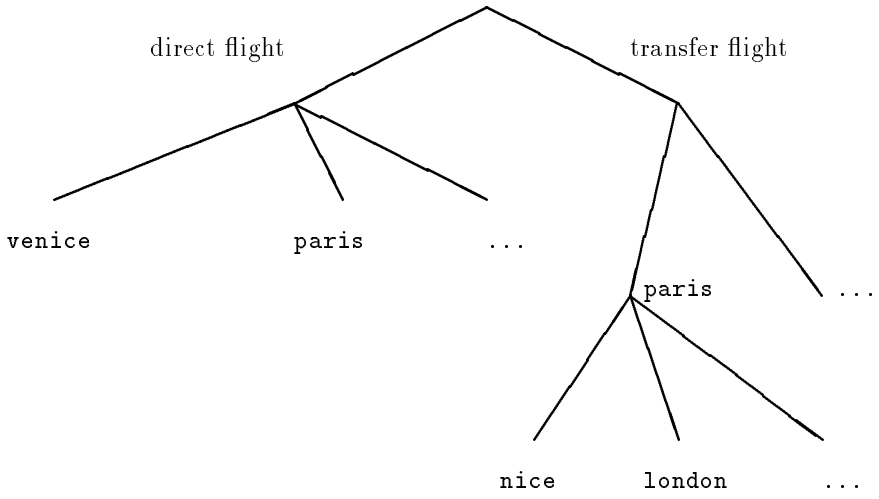

Figure 1.1: The search tree of the holiday destination program

A possible way of exploring or-parallelism in this example is the following. The `destination` predicate can be started by two processors, one exploring the first clause (direct flights), and the other the second clause (transfer flights). The first processor soon creates a choice point for the `flight` predicate, and proceeds down the first branch, starting to execute the `appropriate` goal for the `venice` flight data. While this is done, further processors can join, exploring other choices for the `flight`s. Similarly, the processor working on the second clause for `destination` can be helped by other processors.

This simple program exemplifies the two basic problems to be solved by an or-parallel implementation. First, a variable, such as the destination `City` can be instantiated to different values on different branches of the search tree. This requires a variable binding scheme for keeping track of *multiple bindings*. Second, *scheduling* algorithms have to be devised to associate the processors with the tree branches to be explored. For example, when the first processor finishes the computation of the `appropriate` goal for `City=venice`, it will backtrack to the choice point for `flight`, and may find that exploration of all alternative branches has already been started by other processors. In such a case the scheduling algorithm has to find a choice point with an unexplored branch. This process, together with updating the data structures of the processor necessary for taking up the new branch of the tree, is called *task switching*.

1.1.4 Parallel implementation of logic programming

Research on parallel execution of logic programs was started in the early 1980-s. Much of the initial efforts focused on *stream-parallelism*. In this, the biggest difficulty was caused by trying to combine the parallel execution with Prolog search. This was initially overcome by simply removing the possibility of global search, resulting in the so called committed choice languages. In these languages each clause has to contain a commit pruning operator, which, when reached during execution, kills all the other branches of the procedure. This way the "don't know nondeterminism" of Prolog is replaced by "don't care nondeterminism" of committed choice languages. A detailed survey of committed choice systems can be found in [40].

The first parallel systems aiming to support unrestricted Prolog language appeared at the end of 1980-s. An excellent overview of parallel execution models and their implementations is given in [23]. Here we only briefly survey some of the relevant approaches.

A crucial point in the design of execution models for *independent and-parallelism* is the detection of independence of subgoals. Initial models, such as that of Conery [18], relied on costly run-time checks. DeGroot developed the RAP (Restricted And-Parallel) model [20], in which compile-time analysis is used to simplify the run-time checks needed. A refinement of this approach by Hermenegildo led to the creation of the &-Prolog implementation of independent and-parallelism on shared memory multiprocessors [27].

The Basic Andorra Model [39] was the first practical approach reconciling proper nondeterminism with *dependent and-parallelism*. Here the execution of subgoals continues in and-parallel as long as no choice points are created. This approach was implemented in the Andorra-I system.

About twenty models for *or-parallelism* are listed by [23]. These differ in the way they support the assignment of multiple bindings, and whether they use shared memory or not. Models that do not assume the presence of shared memory rely on either recomputation (the Delphi model of Clocksin [17]), or copying (Conery's closed environments [19], Ali's BC-machine model [2]). The BC-machine model, although first developed for special hardware, was later used for the implementation of the Muse system for shared memory multiprocessors [1].

The early shared memory models, such as the directory tree model [16], the Argonne [11] and PEPSys [3] models, had non-constant variable access time, but relatively little or no task switching overheads. More recent models focused on providing constant-time variable binding access at the expense of potentially non-constant-time task switching[1]. The most developed scheme of this group, the SRI model of D. H. D. Warren [53] forms the basis of the Aurora implementation and is described in more detail in the next section.

Several models and implementations have been developed for exploiting multiple forms of parallelism. Support for both or- and independent and-parallelism is provided by the PEPSys [3], ROPM [29] and ACE [24] models, among others. The combination of dependent and-parallelism with or-parallelism appears in the Basic Andorra Model, and its implementation, Andorra-I. The ambitious Extended Andorra Model [54], which aims to support all three forms of parallelism, has not yet been implemented.

Finally, let us give a brief list of research groups working on parallel logic programming in Hungary. An early and-parallel logic programming implementation was developed by Iván Futó's group in the mid 1980-s. The CS-Prolog (Communicating Sequential Prolog) system supports multiple Prolog threads running concurrently on multi-transputer systems [21]. The group of Péter Kacsuk at the KFKI-MSzKI Laboratory of Parallel and Distributed Systems is working on parallel and distributed Prolog implementations based on dataflow principles [28]. The IQSOFT logic programming group took part in the development and application of the Aurora system.

1.1.5 The Aurora or-parallel Prolog system

Aurora is an implementation of the full Prolog language supporting or-parallel execution of programs on shared memory multiprocessors. It exploits parallelism implicitly, without programmer intervention. It was developed through an informal collaboration, called the Gigalips project, of research groups at the University of Bristol (formerly at the University of Manchester), UK; Argonne National Laboratory (ANL), USA; the Swedish Institute of Computer Science (SICS); and IQSOFT, Hungary (from 1990).

Aurora is based on the SRI model [53]. According to this model the system consists of several *workers* (processes) exploring the search tree of a Prolog program in parallel. Each node of the tree corresponds

[1] In [22] it has been shown that of the three main components of an or-parallel model, the variable access, the task switching, and the creation of environments, at most two can be of constant-time.

to a Prolog choicepoint with a branch associated with each alternative clause. Nodes having at least one unexplored alternative correspond to pieces of *work* a worker can select. Each worker has to perform activities of two basic types:

- executing the actual Prolog code,
- finding work in the tree, providing other workers with work and synchronising with other workers.

The above two kinds of activities have been separated in Aurora: those parts of a worker that execute the Prolog code are called the *engine*, whilst those concerned with the parallel aspects are called the *scheduler*. In the course of development of Aurora, different scheduling techniques have been explored, and several schedulers were developed, such as the Argonne [12], Manchester [13] and Bristol schedulers [6].

The engine component of Aurora is based on SICStus Prolog [15], extended with support for multiple variable bindings. Variable bindings in Prolog can be classified as either unconditional or conditional. In the former case, the binding is made early, before any choice points are made, and so it is shared by all branches. Consequently the unconditional bindings can be stored in the Prolog stacks, as for sequential implementations. For storing the conditional bindings, the SRI model uses *binding arrays*, data structures associated with workers: the Prolog stack stores a binding array index, while the variable value, local to the worker, is stored in the appropriate element of the worker's binding array.

The binding array scheme has a constant-time overhead on variable access. However, task switching involves non-constant-time overhead: the worker has to move from its present node to the node with work, updating its binding array accordingly. The cost of this update is proportional to the length of the path[2]. The scheduler should therefore try to find work as near as possible, to minimise the overheads.

As stated, Aurora supports the *full* Prolog language, including the impure, non-declarative features. Early versions of Aurora provided only the so called *asynchronous* variants of side-effect predicates, which were executed immediately. This meant, for example, that the output predicates were not necessarily executed in the order of the sequential execution.

The final version of Aurora executes the side-effect predicates in the same order as sequential Prolog, as discussed in [26]. This is achieved by *suspending* the side-effect predicate if it is executed by the non-leftmost worker. Suspension means that the worker abandons the given branch of the tree and attempts to find some other work. When the reason for suspension ceases to hold, i.e. when all the workers to the left of the suspended branch have finished their tasks, the branch is *resumed*. Because suspension and resumption has significant overheads, Aurora still provides the "bare" asynchronous predicates, for further experimentation.

Implementing the cut pruning operator in an or-parallel setup poses problems similar to those for the side-effect predicates. A cut operation may be pruned by another cut to its left, hence too early execution of a cut may change the Prolog semantics. Therefore a cut may have to be suspended, if endangered by another cut.

Work in the scope of a pruning operator is called *speculative*, while all other work is called *mandatory*. Parallel exploration of a speculative branch may turn out to be wasteful, if the branch is pruned later. It is an advantage therefore, if the scheduler gives preference to mandatory over speculative work. As pruning is present in all real-life Prolog programs, scheduling speculative work is an important issue.

Detailed discussion of issues related to pruning and speculative work, as well as early work on language extensions, is contained in [25].

1.2 Thesis overview

This section presents a overview of the thesis, showing the problems to be solved, the approach to their solution, the results achieved, and their utilisation.

1.2.1 Problem formulation

The overall goal of the work described in this thesis, as part of a larger research thread, is

[2] More exactly, the cost of the update is proportional to the number of bindings made on the path.

to prove the viability of using shared memory multiprocessors for efficient or-parallel execution of Prolog programs.

This goal is achieved through the development of the Aurora or-parallel Prolog system.

Within this overall goal the problems addressed in the thesis can be classified into three broad areas:

1. **Implementation**: building an or-parallel system supporting the full Prolog language.
2. **Extensions**: extending the Prolog language to support better exploitation of parallelism.
3. **Applications**: prove the usefulness of or-parallel Prolog on large, real-life applications.

We now discuss the specific issues addressed within these areas.

Implementation

As outlined earlier, *scheduling* is one of the crucial aspects of parallel implementations. A scheduler has to keep track of both the workers and the work available. It has to ensure workers are assigned work with as little overhead as possible. To support the full Prolog language, the scheduler has to handle pruning operators, side-effect predicates and speculative work.

In order to choose the best scheduling algorithms, it is important to develop and evaluate multiple schedulers. For this, it is crucial to design an appropriate *interface* between the scheduler and engine components of the parallel system. Development of a proper interface also contributes to the clarification of the issues involved in exploiting parallelism in Prolog.

Evaluation of a parallel Prolog implementation requires appropriate *performance analysis* techniques. The parallel system has to be instrumented to collect performance data and typical benchmarks have to be selected. The gathered data has to be analysed and the main causes of overhead identified. Results of the performance analysis work can contribute to the improvement or re-design of critical system components, e.g. schedulers.

Language extensions

The Prolog language has several impure features, with no declarative interpretation. Language primitives of this kind, such as dynamic data base modification predicates, are quite frequently used in large applications. Although this is often a sign of bad programming style, there are cases where such usage is justified. For example, dynamic predicates can be used in a natural way to implement a continually changing knowledge base.

To support sequential Prolog semantics in a parallel implementation, dynamic predicate updates have to be performed sequentially, in strict left-to-right order. Such restrictions on the execution order, however, involve significant overheads. On the other hand, if asynchronous dynamic predicate handling is used, one is confronted with the usual synchronisation problems due to multiple processes accessing the same memory cell. To solve such problems, *higher level synchronisation* primitives have to be introduced into the parallel Prolog system.

Another reason for using dynamic predicates in Prolog is to enhance its simple search algorithm. For example, optimum search algorithms, such as branch-and-bound and alpha-beta pruning, rely on communication between the branches of the search tree. To extend the search mechanism of Prolog to support such advanced search techniques one is forced to use dynamic predicates, with detrimental effects regarding the exploitation of parallelism. Rather than to come up with ad hoc solutions for particular search problems, it may be advisable to define generic *higher-order predicates for optimum search*, which can be implemented efficiently in a parallel Prolog setup.

Applications

As said earlier, proving the viability of or-parallel Prolog is the main goal of the research strand this thesis is part of. To demonstrate this, one needs Prolog *application problems with abundant or-parallelism*. One

then has to take the Prolog program, normally developed with sequential execution in mind, and transform it in such a way that it produces good speedups, when executed in parallel.

1.2.2 Approach and results

We now discuss how the problems formulated in the previous section were approached, and how their solutions were developed in the context of the Aurora or-parallel Prolog system.

Implementation

In early stages of development of Aurora it became clear that the system has relatively poor speed-ups for certain types of applications. The Manchester scheduler version of Aurora was therefore instrumented to provide various types of profiling information. Both frequency and timing data were collected and main sources of overhead of parallel execution were identified. Special attention was paid to the binding array update overheads associated with the SRI model and to the overheads of synchronisation using locks.

The main conclusion of this *performance analysis* work was that the high cost of task switching in the examined implementation was the main cause of poor speed-ups. The cost of updating the binding arrays, which was feared to be the major cause of overhead, turned out to be insignificant. Similarly, locking costs were found to be acceptably low and there was no major increase in the average locking time when the number of workers was increased.

Based on the experience of the performance analysis work, a new *scheduler*, the so called Bristol scheduler, was developed. It employs a new approach for sharing the work in the Prolog search tree. The distinguishing feature of the approach is that work is shared at the bottom of partially explored branches ("dispatching on bottom-most"). This can be contrasted with the earlier schedulers, such as the Manchester scheduler, which use a "dispatching on topmost" strategy. The new strategy leads to improved performance by reducing the task switching overheads and allowing more efficient scheduling of speculative work.

In parallel with the development of the new scheduler, a new version of the *engine-scheduler interface* was designed. This fundamental revision of the interface was necessitated by several factors. Performance analysis work on Aurora had shown that some unnecessary overheads are caused by design decisions enforced by the interface. Development of the new scheduler and extensions to existing algorithms required that the interface become more general. The Aurora engine was rebuilt on the basis of a new SICStus Prolog version. The interface required extensions to support transfer of information related to pruning operators. Finally, it was decided that an Aurora scheduler was to be used in the Andorra and/or-parallel system, so the interface had to support multiple engines in addition to multiple schedulers.

Language extensions

The problems of *parallel execution of* applications relying on *dynamic predicates* were studied on programs for playing mastermind, a typical problem area using a continually changing knowledge base.

In the case study we first explored some sequential programs for playing mastermind. Subsequently, we considered the problems arising at the introduction of asynchronous database handling predicates. Several versions of the mastermind program were developed, showing the use of various synchronisation techniques. As a conclusion of this work, a proposal for extending Aurora with higher level synchronisation primitives was presented.

The second area of language extensions studied was that of *parallel optimisation*. A general optimum search algorithm was developed, which can be used in the implementation of higher order optimisation predicates. The algorithm covers both the branch-and-bound and the minimax technique, and can be executed efficiently on an or-parallel Prolog system such as Aurora.

Appropriate language extensions were proposed, in the form of new built-in predicates, for embedding the algorithm within a parallel Prolog system. An experimental Aurora implementation of the language extensions using the parallel algorithm was described and evaluated on application examples.

Applications

To prove the viability of or-parallel Prolog, three large search applications were ported to and evaluated on Aurora.

Two search problems were investigated within the area of *computational molecular biology* as experimental Aurora applications: searching DNA for pseudo-knots and searching protein sequences for certain motifs. For both problems 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, with non-uniform memory architecture (NUMA).

First, experiments were performed with the original application code, which was written with sequential execution in mind. For the pseudo-knot program, this also involved adaptation of the low level C code for string traversal[3] to the parallel environment. These results being very promising, further effort was invested in tuning the applications so as to "expose" more parallelism to the system. For this we had to eliminate unnecessary sequential bottlenecks, and reorganise the top level search to permit better load-balancing. Note, however, that the logic of the program was not changed in this tuning process.

The final results of the molecular biology applications were very good. We obtained over 40-fold speedups on the 42-processor supercomputer. This meant converting hours of computation into minutes on scientific problems of real interest.

A third application was examined in the context of the EMRM electronic medical record management system prototype of the CUBIQ project [52]. The medical thesaurus component of EMRM is based on SNOMED (Systematized Nomenclature of Medical Knowledge) [36]. The SNOMED thesaurus contains approximately 40,000 medical phrases arranged into a tree hierarchy. A series of experiments were carried out for *searching* this *large medical knowledge hierarchy*. We used several alternative representation techniques for implementing the SNOMED hierarchy of the EMRM system. Parallel performance of these solutions was measured both on Aurora and on the Muse or-parallel systems.

The experiments have shown that the SNOMED disease hierarchy can be efficiently represented in Prolog using the general frame-extension of the CUBIQ tool-set. Critical points have been highlighted in the implementations, such as the issue of synchronisation at atom construction. When these bottlenecks were avoided, about 90% parallel efficiency could be achieved for six processors in complex searches of the SNOMED hierarchy.

Finally, a new application direction was initiated by work on using Aurora as a vehicle for implementing a *Prolog-based WWW server*. The goal here is to design a single Prolog server capable of interacting simultaneously with multiple clients. This issue is important as AI applications are normally large and slow to start up, so having a separate copy of the application running for each request may not be a viable solution.

We have therefore designed a Prolog server scheme, based on the Aurora or-parallel Prolog system, which allows multiple clients to be executed on a single computer, on a time sharing basis. The solution relies on the capabilities of Aurora to maintain multiple branches of the search tree. Compared with the approach relying on multiple copies of the server application, our solution is characterised by quick start-up and significant reduction in memory requirements. As a further advantage, the single server approach allows easy communication between the program instances serving the different clients, which may be useful e.g. for caching certain common results, collecting statistics, etc.

1.2.3 Utilisation of the results

In this section we discuss the utilisation of the results achieved.

The results of the *performance analysis* work described here served as a basis for practically all subsequent performance measurements of Aurora, such as [33]. The technique used for instrumentation was applied to other Aurora schedulers as well. The set of benchmarks selected was used not only for further Aurora analysis, but also for other or-parallel systems, most notably the Muse [1] system.

The *Bristol scheduler*, the basic design of which is presented here, has evolved to be the main scheduler of Aurora, and is also used in the Andorra-I parallel system [4]. Extending the ideas described here, the Bristol scheduler was further improved with respect to handling speculative work and suspension [8].

[3] The C code was included into the Prolog program through the foreign language interface.

The *engine-scheduler interface* served for implementing the Dharma scheduler [41]. A similar interface was developed for the Muse system as well, see chapter 8 of [35].

The ideas of *language extensions* dealing with dynamic predicates and optimisation were further developed in [9].

The *application prototypes* have proved that Aurora can be used in sizable real-life applications. A *Prolog-based WWW server* approach, similar to the design presented here, has recently been developed independently for the ECLiPSe system [10].

1.3 Structure of the Thesis and contributions

Chapters 2–10 of the thesis contain my main publications in the area of or-parallel logic programming, reproduced here with the kind permission of co-authors. They are grouped into three parts, corresponding to the three research areas described above.

In the sequel I give a brief outline of the research reported on in these publications, and describe my contributions to the work.

1.3.1 Implementation

Chapter 2: **The Aurora or-parallel Prolog system**
Authors: Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumił Hausman
Refereed journal article [31].

This is the main paper on Aurora, written jointly by the three research groups of the Gigalips collaboration. It describes the design and implementation efforts of Aurora as of 1988-89. My contributions to the work described here are in the sections on the Manchester scheduler, on performance analysis and on the Piles application.

Chapter 3: **Performance analysis of the Aurora or-parallel Prolog system**
Author: Péter Szeredi
Refereed conference article [42].

This paper describes the main results of my performance analysis work carried out for the Manchester scheduler version of Aurora. More detailed results are given in the Technical Report [43].

Chapter 4: **Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler**
Authors: Anthony Beaumont, S Muthu Raman, Péter Szeredi, and David H D Warren
Refereed conference article [6].

This paper describes the design and implementation efforts for the first version of the Bristol scheduler. Further details can be found in [5, 7]. My main contribution was the design and initial implementation of the non-speculative scheduling parts of the Bristol scheduler.

Chapter 5: **Interfacing engines and schedulers in or-parallel Prolog systems**
Authors: Péter Szeredi, Mats Carlsson, and Rong Yang
Refereed conference article [49].

This paper gives an outline of the Aurora engine-scheduler interface. The complete description of the interface is contained in reports [48, 14].

I was the principal designer of the interface. I also carried out the implementation of the scheduler side for both the Manchester and Bristol schedulers.

1.3.2 Language extensions

Chapter 6: **Using dynamic predicates in an or-parallel Prolog system**
Author: Péter Szeredi
Refereed conference article [46].

The paper describes the mastermind case study and the language extensions for synchronisation. An earlier version of the paper is available as [44].

Chapter 7: **Exploiting or-parallelism in optimisation problems**
Author: Péter Szeredi
Refereed conference article [47].

This paper describes the optimisation algorithm developed for or-parallel logic programming and the appropriate language extensions. [45] contains an earlier, slightly more elaborate account on this topic.

1.3.3 Applications

Chapter 8: **Applications of the Aurora parallel Prolog system to computational molecular biology**
Authors: Ewing Lusk, Shyam Mudambi, Ross Overbeek, and Péter Szeredi
Refereed conference article [32].

This paper describes the pseudo-knot and protein motif search problems and their solution on Aurora. My main contribution lies in exploring the sequential bottlenecks and transforming the application programs to improve the exploitation of parallelism.

Chapter 9: **Handling large knowledge bases in parallel Prolog**
Authors: Péter Szeredi and Zsuzsa Farkas
Workshop paper [50].

This paper describes the parallelisation of the medical knowledge base application of Aurora. My contribution covers the parallel aspects of the design, and the parallel performance analysis of the application.

Chapter 10: **Serving multiple HTML clients from a Prolog application**
Authors: Péter Szeredi, Katalin Molnár, and Rob Scott
Refereed workshop paper [51].

The paper describes the WWW interface of the EMRM application, the problems encountered during its development, and a design for a multi-client WWW-server application of Aurora. My contribution is the design of the multi-client server.

1.3.4 Summary of publications

Of the nine publications, I am the sole author of three papers (chapters 3, 6, 7). For a further three publications, I am the first author (chapters 5, 9, and 10), reflecting the fact, that I was the principal contributor to the research described.

One of the publications appeared in a refereed journal, six in refereed conference proceedings, and two were presented at workshops.

References

[1] K. A. M. Ali and R. Karlsson. The Muse approach to or-parallel Prolog. *The International Journal of Parallel Programming*, 1990.

[2] Khayri A. M. Ali. OR-Parallel execution of prolog on BC-Machine. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1531–1545, Seatle, 1988. ALP, IEEE, The MIT Press.

[3] U. C. Baron et al. The parallel ECRC Prolog System PEPSys: An overview and evaluation results. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, Tokyo, Japan, November 1988.

[4] Anthony Beaumont, S. Muthu Raman, Vítor Santos Costa, Péter Szeredi, David H. D. Warren, and Rong Yang. Andorra-I: An implementation of the Basic Andorra Model. Technical Report TR-90-21, University of Bristol, Computer Science Department, September 1990. Presented at the Workshop on Parallel Implementation of Languages for Symbolic Computation, University of Oregon, July 1990.

[5] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Scheduling or-parallelism in Aurora with the Bristol scheduler. Technical Report TR-90-04, University of Bristol, Computer Science Department, March 1990.

[6] 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, Lecture Notes in Computer Science, Vol 506, June 1991.

[7] Anthony J. Beaumont. *Scheduling in Or-Parallel Prolog Systems*. PhD thesis, Unversity of Bristol, 1995.

[8] Tony Beaumont and David H. D. Warren. Scheduling Speculative Work in Or-parallel Prolog Systems. In *Logic Programming: Proceedings of the 10th International Conference*. MIT Press, 1993.

[9] Tony Beaumont, David H. D. Warren, and Péter Szeredi. Improving Aurora scheduling. CUBIQ Copernicus project deliverable report, University of Bristol and IQSOFT Ltd., 1995.

[10] Stephane Bressan and Philippe Bonnet. The ECLiPSe-HTTP library. In *Industrial Applications of Prolog*, Tokyo, Japan, November 1996. INAP.

[11] R. Butler, E. Lusk, R. Olson, and Overbeek R. A. ANLWAM: A Parallel Implementation of the Warren Abstract Machine. Internal Report, Argonne National Laboratory, Argonne, IL 60439, 1985.

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

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

[14] Mats Carlsson and Péter Szeredi. The Aurora abstract machine and its emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[15] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1988. SICS Research Report R88007B.

[16] Andrzej Ciepielewski and Seif Haridi. A formal model for or-parallel execution of logic programs. In *IFIP 83 Conference*, pages 299–305. North Holland, 1983.

[17] William Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.

[18] John Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983.

[19] J.S. Conery. Binding environments for parallel logic programs in nonshared memory multiprocessors. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 457–467, San Francisco, August - September 1987. IEEE, Computer Society Press.

[20] Doug DeGroot. Restricted and-parallelism. In Hideo Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[21] Iván Futó. Prolog with communicating processes: From T-Prolog to CSR-Prolog. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 3–17, Budapest, Hungary, 1993. The MIT Press.

[22] Gupta Gopal and Bharat Jayaraman. Optimizing And-Or Parallel implementations. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 605–623. MIT Press, 1990.

[23] Gopal Gupta, Khayri A. M. Ali, Mats Carlsson, and Manuel Hermenegildo. Parallel execution of logic programs: A survey, 1994. Internal report, available by ftp from `ftp.cs.nmsu.edu`.

[24] Gopal Gupta, Manuel Hermenegildo, Enrico Pontelli, and Vítor Santos Costa. ACE: And/Or-parallel Copying-based Execution of logic programs. In Pascal Van Hentenryck, editor, *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 93–109, Massachusetts Institute of Technology, 1994. The MIT Press.

[25] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[26] Bogumił Hausman, Andrzej Ciepielewski, and Alan Calderwood. Cut and side-effects in or-parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[27] Manuel Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 25–39. Springer-Verlag, 1986.

[28] Péter Kacsuk. Distributed data driven Prolog abstract machine (3DPAM). In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 89–118. Wiley & Sons, 1992.

[29] L. V. Kalé. The REDUCE OR process model for parallel evaluation of logic programming. In *Proceedings of the 4th International Conference on Logic Programming*, pages 616–632, 1987.

[30] Robert A. Kowalski. Predicate logic as a programming language. In *Information Processing '74*, pages 569–574. IFIP, North Holland, 1974.

[31] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David 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.

[32] Ewing Lusk, Shyam Mudambi, Ross Overbeek, and Péter Szeredi. Applications of the Aurora parallel Prolog system to computational molecular biology. In Dale Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 353–369. The MIT Press, November 1993.

[33] Shyam Mudambi. Performances of aurora on NUMA machines. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 793–806, Paris, France, 1991. The MIT Press.

[34] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[35] Roland Karlsson. *A High Performance OR-Parallel Prolog System*. PhD thesis, The Royal Institute of Technology, Stockholm, 1992.

[36] D. J. Rothwell, R. A. Cote, J. P. Cordeau, and M. A. Boisvert. Developing a standard data structure for medical language — the SNOMED proposal. In *Proceedings of 17th Annual SCAMC, Washington*, 1993.

[37] P. Roussel. Prolog: Manuel de reference et d'utilisation,. Technical report, Groupe d'Intelligence Artificielle Marseille-Luminy, 1975.

[38] Peter Van Roy, Seif Haridi, and Gert Smolka. An overview of the design of distributed oz. In *Second International Symposium on Parallel Symbolic Computation (PASCO '97)*. ACM Press, July 1997.

[39] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the Eighth International Conference*. The MIT Press, 1991.

[40] Ehud Shapiro. The family of Concurrent Logic Programming Languages. *ACM computing surveys*, 21(3):412–510, 1989.

[41] Raéd Yousef Sindaha. Branch-level scheduling in Aurora: The Dharma scheduler. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 403–419, Vancouver, Canada, 1993. The MIT Press.

[42] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. The MIT Press, October 1989.

[43] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. Technical Report TR-89-14, University of Bristol, 1989.

[44] Péter Szeredi. Using dynamic predicates in Aurora – a case study. Technical Report TR-90-23, University of Bristol, November 1990.

[45] Péter Szeredi. Solving optimisation problems in the Aurora or-parallel Prolog system. In Anthony Beaumont and Gopal Gupta, editors, *Parallel Execution of Logic Programs, Proc. of ICLP'91 Pre-Conf. Workshop*, pages 39–53. Springer-Verlag, Lecture Notes in Computer Science, Vol 569, 1991.

[46] Péter Szeredi. Using dynamic predicates in an or-parallel Prolog system. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Logic Programming Symposium*, pages 355–371. The MIT Press, October 1991.

[47] Péter Szeredi. Exploiting or-parallelism in optimisation problems. In Krzysztof R. Apt, editor, *Logic Programming: Proceedings of the 1992 Joint International Conference and Symposium*, pages 703–716. The MIT Press, November 1992.

[48] Péter Szeredi and Mats Carlsson. The engine–scheduler interface in the Aurora or–parallel Prolog system. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[49] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing engines and schedulers in or-parallel Prolog systems. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, pages 439–453. Springer Verlag, Lecture Notes in Computer Science, Vol 506, June 1991.

[50] Péter Szeredi and Zsuzsa Farkas. Handling large knowledge bases in parallel Prolog. Presented at the Workshop on High Performance Logic Programming Systems, in conjunction with Eighth European Summer School in Logic, Language, and Information, Prague, August 1996.

[51] Péter Szeredi, Katalin Molnár, and Rob Scott. Serving multiple HTML clients from a Prolog application. In Paul Tarau, Andrew Davison, Koen de Bosschere, and Manuel Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications, in conjunction with JICSLP'96, Bonn, Germany*, pages 81–90. COMPULOG-NET, September 1996.

[52] Gábor Umann, Rob Scott, David Dodson, Zsuzsa Farkas, Katalin Molnár, László Péter, and Péter Szeredi. Using graphical tools in the CUBIQ expert system tool-set. In *Proceedings of the Fourth International Conference on the Practical Application of Prolog*, pages 405–422. The Practical Application Company Ltd, April 1996.

[53] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[54] David H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.

Part I

Implementation

Chapter 2

The Aurora Or-Parallel Prolog System[1]

Ewing Lusk
Ralph Butler
Terrence Disz
Robert Olson
Ross Overbeek
Rick Stevens

Argonne[2]

David H. D. Warren
Alan Calderwood
Péter Szeredi[3]

Bristol[4]

Seif Haridi
Per Brand
Mats Carlsson
Andrzej Ciepielewski
Bogumil Hausman

SICS[5]

Abstract

Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors, developed as part of an informal research collaboration known as the "Gigalips Project". It currently runs on Sequent and Encore machines. It has been constructed by adapting Sicstus Prolog, a fast, portable, sequential Prolog system. The techniques for constructing a portable multiprocessor version follow those pioneered in a predecessor system, ANL-WAM. The SRI model was adopted as the means to extend the Sicstus Prolog engine for or-parallel operation. We describe the design and main implementation features of the current Aurora system, and present some experimental results. For a range of benchmarks, Aurora on a 20-processor Sequent Symmetry is 4 to 7 times faster than Quintus Prolog on a Sun 3/75. Good performance is also reported on some large-scale Prolog applications.

2.1 Introduction

In the last few years, parallel computers have started to emerge commercially, and it seems likely that such machines will rapidly become the most cost-effective source of computing power. However, developing parallel algorithms is currently very difficult. This is a major obstacle to the widespread acceptance of parallel computers.

Logic programming, because of the parallelism *implicit* in the evaluation of logical expressions, in principle relieves the programmer of the burden of managing parallelism explicitly. Logic programming therefore offers the potential to make parallel computers no harder to program than sequential ones, and to allow software to be migrated transparently between sequential and parallel machines.

[1]This paper has appeared in *New Generation Computing* **7** (1990) [20]
[2]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.
[3]*On leave from* SZKI, Donáti u. 35-45, Budapest, Hungary
[4]Department of Computer Science, University of Bristol, Bristol BS8 1TR, U.K. *The group was previously at:* Department of Computer Science, University of Manchester, Manchester M13 9PL, U.K.
[5]Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden

It only remains to determine whether a logic programming system coupled with suitable parallel hardware can realise this potential. The Aurora system is a first step towards this goal. Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors. It currently runs on Sequent and Encore machines. It has been developed as part of an informal research collaboration known as the "Gigalips Project".

The Aurora system has two purposes. Firstly, it is intended to be a research tool for gaining understanding of what is needed in a parallel logic programming system. In particular, it is a vehicle for making concrete an abstract parallel execution model, the SRI model, in order to evaluate and refine it. The intention is to evaluate the model not only on the present hardware, but also to look towards possible future hardware (not necessarily based on shared physical memory).

Secondly, Aurora is intended to be a demonstration system, that will enable experience to be gained of running large applications in parallel. For this purpose, it is vital that the system should perform well on the present hardware, and that it should be a complete and practical system to use.

In order to support *real* applications efficiently and elegantly, it is necessary to implement a logic programming language that is at least as powerful and practical as Prolog. The simplest way to ensure this, and at the same time to make it easy to port existing Prolog applications and systems software, is to include full Prolog with its standard semantics as a true subset of the language. This we have taken some pains to achieve.

The bottom line for evaluating a parallel system is whether it is truly competitive with the best sequential systems. To achieve competitiveness, it is necessary to make a parallel logic programming system with a single processor execution speed as close as possible to state-of-the-art sequential Prolog systems, while allowing multiple processors to exploit parallelism with the minimum of overhead. This has been our goal in Aurora.

To summarise the objectives towards which Aurora is addressed, they are to obtain truly competitive performance on real applications by transparently exploiting parallelism in a logic programming language that includes Prolog as a true subset.

In this paper, we discuss the issues that must be confronted in or-parallel Prolog implementation, and describe the design and main implementation features of the current Aurora system. We present some experimental results illustrating the performance of the system on a number of benchmarks, and also report our experience of porting a number of large-scale applications to Aurora. We conclude by summarising the current state of Aurora and outlining directions for further research.

2.2 Background

In this section we describe the setting in which Aurora was developed and give a short history of the Gigalips Project.

2.2.1 Sequential Prolog Implementations

Prolog implementation entered a new era when the first compiler was introduced, for the DEC-10 [26]. The speed of this implementation, and the portability and availability of its descendant, C-Prolog, set a language standard, now usually referred to as the "Edinburgh Prolog". The DEC-10 compilation techniques led as well to a standard implementation strategy, usually called the WAM (Warren Abstract Machine) [25]. In a WAM-based implementation, Prolog source code is compiled into the machine language of a stack-based abstract machine. A portable emulator of this abstract machine (typically written in C) yields a fast, portable Prolog system, and a non-portable implementation of crucial parts of the emulator can increase speed still further. A parallel implementation of Prolog is achieved by parallelising this emulator.

There are now many high-quality commercial and non-commercial Prolog systems based on the WAM. A parallel implementation can obtain considerable leverage by utilising an existing high-quality implementation as its foundation. We use the Sicstus [6, 5] implementation, one of the fastest portable implementations.

Using a fast implementation is important for two reasons. Firstly, the single most important factor determining the speed of a parallel version is the speed of the underlying sequential implementation. Secondly, many research issues related purely to multiprocessing only become apparent in the presence of a fast sequential implementation. (Speedups are too easy to get when speed is too low).

2.2.2 Multiprocessors

It is only in the last few years that multiprocessors have emerged from the computer science laboratories to become viable commercial products marketed worldwide. Startup companies like Sequent, Encore, and Alliant have made shared-memory multiprocessors commonplace in industry and universities alike. Such machines are relatively inexpensive compared with comparable mainframes, and provide a standard operating environment (Unix^{TM}) making them extremely popular as general-purpose computation servers. A similar revolution is happening with local-memory multiprocessors, sometimes called "multicomputers", but these are currently more specialised machines, despite their scalability advantages.

What the new breed of machines does *not* provide is a unified way of expressing and controlling parallelism. A variety of compiler directives and libraries are offered by the vendors, and while they do allow the programmer to write parallel programs for each machine, they provide neither syntactic nor conceptual portability. A number of researchers are developing tools to address these issues, but at a relatively low level (roughly the same level as the language they are embedded in, such as C or Fortran). A goal of the Gigalips Project is to demonstrate the effectiveness of logic programming as a vehicle for exploiting parallelism on these machines.

2.2.3 Or-Parallelism

As is well known, there are two main kinds of parallelism in logic programs, and-parallelism and or-parallelism. The issues raised in attempting to exploit the two kinds of parallelism are sufficiently different that most research efforts are focussing primarily on one or the other. Much early and current work has been directed towards and-parallelism, particularly within the context of "committed choice" languages (Parlog, Concurrent Prolog, Guarded Horn Clauses) [13, 23]. These languages exploit **dependent** and-parallelism, in which there may be dependencies between and-parallel goals. Other work [10, 18] has been directed towards the important special case of **independent** and-parallelism, where and-parallel goals can be executed completely independently.

The committed choice languages have been viewed primarily as a means of expressing parallelism *explicitly*, by modelling communicating processes. In contrast, one of our main goals is to exploit parallelism *implicitly*, in a way that need have little impact on the programmer. This viewpoint has led us to take a rather different approach, and to focus in particular on or-parallelism.

There are several reasons for focussing on or-parallelism as a first step. Briefly, in the short term, or-parallelism seems easier and more productive to exploit transparently than and-parallelism. However, none of these reasons precludes integrating and-parallelism at a later stage, and indeed this is precisely the goal of current work on the Andorra model and language [14, 31]. The advantages of or-parallelism are:

- **Generality.** It is relatively straightforward to exploit or-parallelism without restricting the power of the logic programming language. In particular, we retain the ability we have in Prolog to generate all solutions to a goal.
- **Simplicity.** It is possible to exploit or-parallelism without requiring any extra programmer annotation or complex compile-time analysis.
- **Closeness to Prolog.** It is possible to exploit or-parallelism with an execution model that is very close to that of sequential Prolog. This means that one can take full advantage of existing implementation technology to achieve a high absolute speed per processor, and also makes it easier to preserve the same language semantics.
- **Granularity.** Or-parallelism has the potential, at least for a large class of Prolog programs, of defining large-grain parallelism. Roughly speaking, the *grain size* of a parallel computation refers to the amount of work can be performed without interaction with other pieces of work proceeding in parallel. It is much easier to exploit parallelism effectively when the granularity is large.
- **Applications.** Significant or-parallelism occurs across a wide range of applications, especially in the general area of artificial intelligence. It manifests itself in any kind of search process, whether it be exercising the rules of an expert system, proving a theorem, parsing a natural language sentence, or answering a database query.

2.2.4 Issues in Or-Parallel Prolog Implementation and Early Work

The main problem with implementing or-parallelism is how to represent different bindings of the same variable corresponding to different branches of the search space. The challenge is to do this in such a way that the overhead of binding, unbinding and dereferencing variables is kept to a minimum compared with fast sequential implementations. Various or-parallel models have been proposed [27, 17, 30, 1, 9], incorporating different binding schemes.

An early binding scheme was that of the SRI model, first suggested informally by Warren in 1983 and subsequently refined [28]. The early form of this model partly influenced Lusk and Overbeek in the design of the pioneering system, ANL-WAM [12], one of the first or-parallel systems to be implemented. However, they ended up implementing an alternative, rather more complex, binding scheme.

ANL-WAM was first implemented on the Denelcor HEP and later ported to other shared-memory machines. It demonstrated that good speedups could be obtained on Prolog programs, but suffered from the fact that the quality of its compiler and emulator were well behind the state of the art. Also there were considerable overheads associated with the binding scheme and treatment of parallel choicepoints. However, ANL-WAM provided a concrete demonstration of what could be achieved, and was a major inspiration behind the formation of the Gigalips Project. The experience of ANL-WAM, together with that from early work on or-parallelism in Sweden [7, 8, 17], has led to the refined version of the SRI model that has now been implemented in Aurora.

2.2.5 A Short History of the Gigalips Project

At the Third International Conference on Logic Programming in London in the summer of 1986, a meeting was held of representatives of several groups interested in various aspects of parallelism in logic programming. It was agreed that there would be a core development project, open to participation by anyone, and that anyone with related research interests was welcome to stay in close contact. Over the next year the project became known as the Gigalips Project, and the core development centered on the Aurora system described in this paper. The implementors were groups from Argonne National Laboratory, the University of Manchester, and the Swedish Institute of Computer Science. The Manchester group subsequently moved to the University of Bristol in the summer of 1988. Beginning in the spring of 1987, gatherings of the key participants were held approximately every three months to decide on major issues and merge work that had been done locally. Also attending these gatherings were researchers from ECRC, Imperial College, MCC, Stanford and elsewhere. As a result, the Gigalips Project has been not only a design and implementation effort, but also a medium for pursuing common research interests in parallel logic programming systems.

2.3 Design

Aurora is based on the SRI model, and most of the design decisions are as described in an earlier paper [28]. In this section, we summarise the main features of the design, emphasising those aspects which are not covered in the earlier paper.

2.3.1 The Basic SRI Model

In the SRI model, a group of **workers**[6] cooperate to explore a Prolog **search tree**, starting at the root (the topmost point). The tree is defined implicitly by the program, and needs to be constructed explicitly (and eventually discarded) during the course of the exploration. Thus the first worker to enter a branch constructs it, and the last worker to leave a branch discards it. The actions of constructing and discarding branches are considered to be the real **work**, and correspond to ordinary resolution and backtracking in Prolog. When a worker has finished one continuous piece of work, called a **task**, it moves over the tree to take up another task. This process is called task switching or **scheduling**. Workers try to maximise the time they spend working and minimise the time they spend scheduling. When a worker is working, it adopts a depth-first left-to-right search strategy as in Prolog.

[6] A worker is an abstract processing agent. We use this term in order to leave unspecified the relationships with hardware processors and operating system processes.

The search tree is represented by data structures very similar to those of a standard Prolog system such as the WAM. Workers that have gone down the same branch share data on that branch. As soon as data becomes potentially shareable through creation of a choicepoint, it may not be modified. To circumvent this restriction, each worker has a private **binding array**, in which it records **conditional bindings**, i.e. bindings to variables which have become shareable. The binding array gives immediate access to the binding of a variable. Conditional bindings are also recorded chronologically in a shareable binding list called the **trail** (similar to that in the WAM). Unconditional bindings are implemented as in the WAM by updating the variable value cell; they do not need to be recorded in the trail or binding array.

Using the binding array and trail, the basic Prolog operations of binding, unbinding, and dereferencing are performed with very little overhead relative to sequential execution (and remain fast, *constant-time* operations). The binding array introduces a significant overhead only when a worker switches tasks. The worker then has to update its binding array by deinstalling bindings as it moves up the tree and installing bindings as it moves down the tree, always keeping its binding array in step with the trail.

The major advantage of the SRI model, compared with other models [27, 12, 17], is that it imposes minimal overhead on a worker while it is working.

2.3.2 Extending the WAM

We will now describe in general terms how the SRI model has been implemented as an extension to the WAM. An important design criterion has been to allow any choicepoint to be a candidate for or-parallel execution.

The nodes of the search tree correspond to WAM choicepoints, with a number of extra fields to enable workers to move around the tree and to support scheduling generally. The extra fields depend on the scheduling scheme, but typically include pointers to the node's parent, first child node and next sibling nodes, and a lock. Most of these extra fields do not need to be initialised, and can be ignored, until the node is made **public**, i.e. accessible to other workers. This will be explained in more detail shortly. Most other WAM data structures are unchanged. However trail entries contain a value as well as a variable address, environments acquire an extra field, and choicepoints acquire a further two fields to support the binding array.

Each worker maintains a binding array to record its conditional bindings. A value cell of a variable that is not unconditionally bound contains an offset that identifies the corresponding location in the binding array where the value, if any, is to be found. When a variable is initialised to unbound, it is allocated the next free location in the binding array. Having unbound variables initialised to such offsets simplifies the testing of seniority that is necessary when one variable is bound to another.

In our implementation, there is one worker per operating system process, and each process has a separate address space which may be only partially shared with other processes. We take advantage of this by locating all binding arrays at a fixed address in unshared virtual memory. This means that workers can address their binding arrays directly rather than via a register, and that binding array offsets in variable value cells can be actual addresses.

The binding array is divided into two parts: the **local binding array** and the **global binding array**, corresponding to variables in, respectively, the WAM (local) stack and heap (or global stack). Each part of the binding array behaves as a stack growing and contracting in unison with the corresponding WAM area. The worker maintains a register to keep track of the top of the global binding array. The need to access a similar register for the local binding array is avoided by performing most of the allocation process at compile-time (see later).

2.3.3 Memory Management

To support the or-parallel model, the WAM stacks need to be generalised to "cactus stacks" mirroring the shape of the search tree.

To achieve this, each worker is allocated a segment of virtual memory, divided into four **physical** stacks: a **node stack**, an **environment stack**, a **term stack**, and a **trail**. The first two correspond to the WAM (local) stack unravelled into its two parts, and the second two correspond to the WAM heap and trail respectively.

Each worker always allocates objects in its own physical stacks, but the objects themselves may be linked

(explicitly or implicitly) back to objects in other workers' stacks forming a **logical** stack.

The main difference from the WAM arises when a worker needs to switch tasks. At a task switch the worker may need to preserve data at the base of its stacks for the benefit of other workers. In this case, data for the new task will be allocated on the stacks after the old data. If any of the old data later becomes unneeded, "holes" will appear in the stack. These holes will be tolerated until reclaimed by an extension of the normal stack mechanism. The holes correspond to **ghost nodes**, i.e. nodes which have been marked as logically discarded by the last worker to need them, but which have not yet been physically removed from memory. A ghost node and the associated "holes" in the other stacks will be reclaimed when the worker who created them finds the ghost node at the top of its node stack. This occurs at task switching.

Regarding two possible optimisations mentioned in the earlier paper on the SRI model [28], the present Aurora implementation does not perform **promotion of bindings**, and **straightening** has only been implemented in an experimental form (in the Manchester scheduler, described later).

2.3.4 Public and Private Nodes

We have already mentioned the distinction between public and private nodes. It has the effect that the search tree is divided into two parts: an upper, public, part accessible to all workers, and a lower, private, part each branch of which is only accessible to the worker that is creating it. This division has two purposes:

- It enables a worker working in the private part of the tree to behave very much as a standard sequential engine, without being concerned about locking or maintaining the extra data in the tree needed for scheduling purposes.
- It provides a mechanism by which the granularity of the exploited or-parallelism can be controlled. By keeping work private, a worker can prevent its tasks from becoming too fragmented.

We think of the worker as having two personas: a scheduler and an engine. When the worker enters the public part of the tree, it becomes a scheduler, responsible for the complexities of moving around the public part of the tree and coordinating with other workers. When the worker enters the private part of the tree, it becomes an engine, responsible for executing work as fast as possible. Periodically, the engine pauses to perform various scheduling functions, the chief one of which is to make its topmost private node public if necessary. The frequency with which nodes are allowed to be made public provides the granularity control mentioned.

To maintain the integrity of the public part of the tree, it is necessary for a (busy) worker always to have a topmost private node for the public node above it to point to. This private node has a special status, in that typically it must have a lock and sibling and parent pointers, amongst other things. It is called a **sentry node**.

In the initial implementation of Aurora, a **dummy node** was created when a worker was launched on a new task to serve as the sentry node. This simplified the adaptation of the existing engine, but resulted in the search tree becoming cluttered with superfluous dummy nodes. We have now implemented the concept of an **embryonic node** as originally described [28]. The embryonic node is "fleshed out" by the engine when it needs to create a choicepoint. The implementation of embryonic nodes involved separating the fields of a node into two parts, the scheduler part and the engine part, with a pointer from the former to the latter. This separation was necessary because a WAM choicepoint is not of a fixed size but varies according to the arity of the predicate.

2.3.5 Scheduling

The function of the scheduler is to rapidly match idle workers with available work. Principal sources of overhead that arise and need to be minimised include installation and deinstallation of bindings, locking to control access to shared parts of the search tree, and performing the bookkeeping necessary to make work publicly accessible. In addition, one wants the scheduler to prefer "good" work, for example larger grain size computations or less speculative ones. (Work is said to be **speculative** if it may be pruned, i.e. become unnecessary, due to a cut or commit).

What makes the scheduling problem interesting is that these goals are not always compatible. For example, large-grain work may become available far away in the tree, while smaller-grain or speculative work is

available nearby. It is not clear what to do with idle workers when there is (temporarily) no work available for them. They can stay where they are or try to guess where work will appear next and position themselves nearby. Movement to work is over unstable terrain, since the tree is constantly being changed by other workers, and so a way must be found to navigate through it with as little locking as possible. Scheduling is also complicated by cut, commit, and suspension (see below). Finally, a scheduling algorithm that works well on a particular class of programs is likely to perform poorly on a different class, so that compromises are inherent.

Because scheduling is such an open research problem, we have experimented with a number of alternative schemes within Aurora. Three quite distinct schemes have been implemented and will be described in a later section.

2.3.6 Cut, Commit, Side Effects and Suspension

Aurora supports **cut** and **cavalier commit**. Cut has a semantics strictly compatible with sequential Prolog. It prunes branches to the right of the cutting branch in such a way that side effects (including other cuts) are prevented from occurring on the pruned branches. Cavalier commit is a relaxation of cut that prunes branches both to the left and right of the cutting branch, and is not guaranteed to prevent side effects from occurring on the pruned branches. Cut selects the first branch through a prunable region; commit selects any one branch through a prunable region.

Cut is currently implemented by requiring it to suspend until it is the leftmost branch within the subtree it affects. This is the simplest but by no means the most efficient approach. Recent improvements [15] require cut to suspend only so long as it could possibly be pruned by cuts with smaller scopes. Cavalier commit is more straightforward to implement in that it doesn't require any suspension mechanism.

Aurora also supports standard Prolog built-in predicates including those which produce side effects. Calls to such predicates are required to suspend until they are on the leftmost branch of the entire tree. We have also implemented "cavalier" (or "asynchronous") versions of certain predicates, which do not require any suspension [16].

2.3.7 Other Language Issues

The current implementation supports some interim program annotation to control parallelism. If the declaration:

```
:- sequential <procedure>/<arity>.
```

is included in a source file, then the or-branches of `<procedure>/<arity>` cannot be explored in parallel. Thus a programmer currently identifies predicates whose clauses must be executed sequentially. The compiler and emulator are then able to mark choicepoints according to whether or not they can be explored in parallel.

All the predicates in a file may be declared sequential by placing a declaration:

```
:- sequential.
```

at the head of the file. This may be overridden for individual predicates by declaring them parallel (using analogous syntax).

Sequential declarations were introduced as an interim measure before cut and side effects were properly supported. At that time cut behaved as a true cut in sequential code but as a commit in parallel code. Now cut and side effects are correctly supported. Sequential declarations are still available to the programmer as a means to restrict the parallelism that is exploited. For non-speculative work, there appears to be little point in restricting the parallelism. For speculative work, however, the present schedulers do not have an adequate strategy, and there is therefore currently scope for the programmer to usefully restrict the parallelism [2].

2.4 Implementation

The implementation of Aurora is based on Sicstus Prolog combined with the or-parallel implementation framework developed for ANL-WAM. The system is intended to provide a framework within which various implementation ideas could be tried out. These two factors have led to a structure for Aurora consisting of a number of identifiable components, each relatively independent of the others. The main components are the engine and scheduler.

A clean interface between the engine and the scheduler has been defined and implemented [3]. It defines the services that the engine must provide to the scheduler and those that the scheduler provides to the engine. This interface allows different engines or schedulers to be inserted into the system with the minimum of effort. A scheduler testbed, compatible with the interface, allows different schedulers to be tested on simulated search trees in isolation from the full system. This is an invaluable aid to debugging scheduling code.

2.4.1 Prolog Engine

The foundation of Aurora is Sicstus Prolog [6, 5], a relatively complete Prolog system implemented in C, which has been ported to a wide range of Unix machines. Aurora is currently based on version 0.3 of Sicstus, although migration to version 0.6 is underway. Sicstus comprises a compiler, emulator, and run-time system. The most basic component is the emulator or engine. The Sicstus engine is a C implementation of the WAM with certain extensions, including the ability to delay goals (by wait declarations). Choicepoints and environments are kept in separate stacks, which turns out to be essential for the SRI model. To produce a parallel version of the engine supporting the SRI model, a number of changes had to be made. The total performance degradation as a result of these changes has been found to be around 25% (see later).

2.4.1.1 Cactus Stack Maintenance

Each worker maintains the boundary between the public and private sections of its node stack in a **boundary** register which points to the youngest public node. This governs what part of the node stack has to be kept for the benefit of other workers. Fields of the youngest public node define the boundaries for the other stacks and for the binding arrays. When a task is started, the boundary is moved back over zero or more ghost nodes, thus shrinking the public section. The boundary register is updated as the engine makes work public (see below). It is also used to detect on backtracking when to leave the engine.

2.4.1.2 Handling of Variable Bindings

Adapting the standard WAM for the SRI model binding scheme implies a number of changes. Unbound or conditionally bound variables are represented as **binding array references**, i.e. as pointers into a binding array, marked with a special tag. The corresponding array location is initialised to `UNBOUND`. Other values indicate that the variable has been bound. When accessing a variable or an argument of a structure, one has to cater for the possibility of encountering a binding array reference, in which case one has to access the binding array. Seniority tests (for variable-variable bindings and for testing whether variable bindings need to be trailed) are performed by comparing binding array references, rather than variable addresses.

For the term stack, a new WAM register maintains the next available binding array reference, and is incremented for each new variable. The situation is somewhat different for variables in the environment stack, as explained in the following section. Choicepoints acquire two new fields to record the tops of the binding arrays.

2.4.1.3 The Environment Stack

Allocating binding array slots for variables in the environment stack is performed at compile time, in contrast to the mechanism described above for the term stack. This is done by storing in each environment a base pointer into the local binding array, denoted **CL(E)**, and extending two WAM instructions with an extra argument:

`call(P,n,j)`
: Call procedure P with n permanent variables still to be used, j out of these having been allocated in the local binding array by `put_variable`. The n and j operands are denoted `EnvSize(I)` and `VarCount(I)`, respectively.

`put_variable(Yn,Ai,j)`
: Set A_i to reference the new unbound variable Y_n whose binding array reference is computed as j + the base pointer stored in the environment.

The algorithm to compute **A**, the top of environment stack, is extended to also compute **LV**, the top of local binding array. If the current environment is younger than the current choicepoint, then **A** is **E** + `EnvSize`(**CP**) (as usual), and **LV** is **CL(E)** + `VarCount`(**CP**). Otherwise **LV** is the top of local binding array field of **B**, and **A** is the top of environment stack field of **Bp**. Here **Bp** is a new WAM register, denoting the youngest choicepoint in the worker's *own* node stack. It is usually different from **B** (the current choicepoint) only when a task is started; as soon as a choicepoint is created, **B** and **Bp** get the same value. When adjusting **B**, **Bp** has to be recomputed as well. However, this overhead was judged worthwhile as it speeds up the computation of **A** which occurs more frequently than updates of **B**.

The base pointer field **CL(E)** also serves as an indicator of the age of an environment. This proves useful when comparing ages of choicepoints and environments, as address comparisons cannot be used. The compiler ensures that the chain of base pointers form a strictly increasing sequence for this comparison to work.

2.4.1.4 Cut and Cavalier Commit

After a cut or commit operation which resets the current choicepoint to an earlier value N, it becomes mandatory to tidy the portion of the trail which is younger than N. Tidying means to reprocess all bindings which were recorded earlier as conditional and make them unconditional where appropriate. If this is not done, there might be garbage references in the trail to a portion of the environment stack which is being reused by tail recursion optimisation. It is a property of the SRI model that a trailed item always refers to a variable whose value is a binding array reference. This property might be violated if the trail is not tidied, with fatal effects when attempting to reset non-existent variables.

The cut/commit operation must also treat cutting within the private section and cutting into the public section as two separate cases, and call a scheduler function to perform the latter. In the latter case, the scheduler may refuse to perform the cut, in which case the engine suspends as described in the following section. If the scheduler does perform the cut it may order other workers to abort their current tasks.

To support suspension of cuts, the compiler provides extra information about what temporary variables need to be saved until the suspended task is resumed. This extra information also encodes the distinction between a cut and a cavalier commit.

2.4.1.5 A Suspension Mechanism

An ability was added to suspend work until the current branch of the computation tree is the left-most one, either globally or with respect to some ancestral node. The global suspension test was added to all built-in side effect predicates. The local test is used for cuts (see above).

To suspend work, the engine pushes a node with a single alternative denoting the current continuation, makes the entire private section public[7], and returns control to the scheduler. It is up to the scheduler to decide when the suspended work may be resumed.

2.4.1.6 Other Multiprocessing Issues

A mechanism was added to allow the engine to periodically perform certain scheduling functions, notably to make work public or to abort the current task. At every procedure call, a counter controlling the granularity is decremented to determine whether to seek to perform such action.

[7] This is avoided in the new version of Aurora currently being implemented.

Access to certain global data structures (symbol tables, predicate databases etc.) had to be synchronised by using locks. Currently each worker performs input/output, although this would probably be better handled by a dedicated Unix process to avoid multiple accesses to buffers and control blocks.

Special support for concurrent executions of `setof` has been provided. In the Aurora implementation of `setof(X,P,L)`, each invocation acquires its own **save area**, where instances of `X` are saved. Each such save area is itself serialised by a lock, to cater for parallelism within `P`.

2.4.2 Schedulers

Scheduling issues are an active area of research within the project, and the engine/scheduler interface allows us to experiment with different alternatives. To date four quite distinct schedulers have been implemented, an early interim solution and three more recent and more complete solutions: the Manchester scheduler, the Argonne scheduler, and the Wavefront scheduler. These are described below. The Wavefront scheduler was the last to be developed and has only recently become fully functional.

The earliest scheduler was based on a strategy described by SICS [17]. The implementation was modeled loosely on ANL-WAM and featured a global scheduling mechanism. That is, a single lock protected the data structures necessary to determine what branch of the tree an idle worker would explore next. It was anticipated that this global lock would represent a bottleneck as machines with more processors become available. The later schedulers use a more local scheme for assigning available work to available workers.

The three current schedulers are very similar in their level of completeness, all handling cut, commit and sequential side effect predicates correctly. Moreover, although they have rather different ways of implementing their responsibilities, they do share a number of strategy decisions. All three schedulers release work only from the topmost node on a branch. This may be regarded as a breadth-first strategy and is a simple attempt to maximise the size of tasks for their engines. In general the schedulers attempt to maintain one, live, shareable node on their current branch, irrespective of whether any other worker is currently idle (although the Manchester scheduler has relaxed this requirement with its "lazy release" mechanism). In general, if a cut or side effect predicate cannot be executed due to its not being on the leftmost branch in the appropriate subtree then the schedulers suspend that work, freeing the worker to look for another task. None of the schedulers currently gives special treatment to speculative work: all work is regarded as being equally worthwhile; however better treatments of speculative work are being developed.

2.4.2.1 The Manchester Scheduler

The aim of the Manchester scheduler [4] is to match workers to available work as well as possible. When there are workers idle, any new piece of shareable work is given directly to the one judged to be closest in the search tree. Conversely, when a worker finishes a task it attempts to claim the nearest piece of available work; if none exists, it becomes idle at its current node in the tree.

The matching mechanism relies upon each worker having a unique number and there being a worker map in each node indicating which workers are at or below the node. There are, in addition, two global arrays, both indexed on worker number. One array indicates the work each worker has available for sharing and its migration cost, and the other indicates the status of each worker and its migration cost if it is idle. The migration cost of a node is taken to be the number of trail entries from the root down to that node. If a worker is looking for work, then by examining the bit map in its current node it knows which work array entries need be considered and it can choose the one with the lowest migration cost. If the subtree contains no shareable work then scanning up the branch towards the root allows progressively larger subtrees to be considered. The worker status array allows the use of an analogous procedure when determining the best idle worker to hand work to.

The execution of cuts and commits also relies on the bitmaps to locate and notify the workers in the branches to be pruned. In general, the task of cleaning up the pruned subtree is left to the workers being cut, so that the cutter can proceed with its own work.

A number of refinements to the basic scheduling algorithm have been introduced into the Manchester scheduler.

Shadowing. Idle workers try to distribute themselves evenly over the tree, each shadowing an active (working) worker, in the hope that it will release some work later on.

Delayed re-release. When a piece of work is acquired by a worker from a node, release of further alternatives from the same node is disabled for a short period of time. Since it is not known in advance how big a task will be, it was thought that delayed re-release would lead to a better distribution of work and help avoiding congestion of workers.

Lazy release. Nodes are made public only when there are idle workers waiting for work. This way one can avoid creating public nodes that will never be shared. A disadvantage of this scheme is that when a worker runs out of work it must first become idle and wait till the others notice this fact before it can get hold of a piece of work.

Straightening. This operation, actually defined by the basic SRI model [28], removes a dead node when a worker dies back to it leaving just one other branch. The structure of the tree can be simplified considerably and all the future movements of workers through the given branch can benefit from the straightening. Note that the promotion of bindings has not been incorporated into the straightening operation in the current implementation.

Of the above refinements shadowing and lazy release have been shown to be the most useful [24], while delayed re-release proved to be detrimental for most of the examples. Straightening leads to little improvement in the speed, probably because the actual implementation of this operation is quite complex and also because it causes a noticable increase in overall congestion for locks. We believe that the implementation of straightening can be improved, so that it will have a beneficial effect on overall execution time.

2.4.2.2 The Argonne Scheduler

The philosophy of the Argonne scheduler [2] is to allow workers to make local decisions; very little use is made of global data. Any worker that is in the public part of the tree is positioned at some particular node. In order to find work to do, it makes a decision about whether to choose an alternative at its current node (if there is one) or to move along an arc of the tree to a nearby node and repeat the decision process. This local decision and one-step-at-a-time movement leads to an easily modifiable scheduling strategy.

Data to support this strategy is local. A bit in each node indicates whether or not an unexplored alternative exists at this node or below. These bits attract workers from above. Workers are "eager" in the sense that as soon as they become available they begin an active search for work. Only when they believe they are optimally positioned to take advantage of new work that might appear do they become inactive.

The current strategy is to try to maintain at least one active public node on each branch of the tree. Thus the scheduler takes a liberal approach to releasing work, and workers are correspondingly eager in their pursuit of it, trying to position themselves where work might appear even if no work is currently available. The potential drawbacks of these decisions are that nodes may become public that are never actually shared, thus needlessly increasing the overhead of claiming an alternative from the node. Workers in eager pursuit of potential work may move away from places in the tree where work is just about to appear, so that it would have been better not to be so "eager". The impact of these drawbacks depends in general on the particular Prolog program being executed.

2.4.2.3 The Wavefront Scheduler

From the viewpoint of scheduling, the most interesting part of the search tree is at the boundary between public and private regions. In particular, assuming a topmost node scheduling strategy, new work is found only at the youngest public nodes. The basic idea behind the Wavefront scheduler is to link together these "interesting" nodes allowing for a more direct access to work. We call this linked chain of nodes the **wavefront**.

The most important property of the wavefront is that all active public nodes are to be found there. Such nodes may have any number of children, that is, workers that have taken an alternative and are privately exploring it. The set of children is called the **wavelet**, and can be seen as representing a possible future expansion of the wavefront. When a worker takes the last alternative, linking itself in as the last child in the wavelet, the wavefront is expanded downwards into the wavelet. When a worker fails up to its public-private boundary it looks for new work. In the case when the worker is in a wavelet, finding new work is trivial, as the parent node is still active; the next alternative is simply taken and the sentry node (see Section 2.3.4)

is moved to its new position as the last child within the wavelet. If the worker is on the wavefront, on the other hand, the worker scans the wavefront for work.

When a worker moves from one position in the tree to another, it must update its binding array accordingly, using the trail. In the other schedulers, workers actually move through the tree node by node updating binding arrays on the way. In the Wavefront scheduler, the wavefront provides not only a more direct way of finding work, but also the information necessary for updating binding arrays. Nodes are augmented with a new field, called the join, which, although by necessity being placed in one node, actually defines a relation between two neighbouring wavefront nodes. The join is a pointer to the lowest node that a wavefront node has in common with its right neighbour. Updating the binding array to reflect a move between neighbouring nodes is then done by using the logical trail: from the first wavefront node to the join node, to deinstall bindings, and from the second wavefront node to the join node, to install bindings.

In the Wavefront scheduler, a worker that becomes idle keeps its position in the wavefront while looking for work. The lowest of its two joins defines the region in which it is alone. An idle worker periodically checks this region, reclaims memory, and grabs sequential work, if any. An idle worker periodically looks for work along the wavefront. When work is found, the worker (1) reserves the work, linking itself into the wavelet as the last child, (2) possibly performs a wavefront expansion and installs/deinstalls bindings as appropriate for its new position, and (3) removes itself from its previous wavefront position. Removing a node from the wavefront is simple: at most one of the two joins associated with the node has to be updated (the lowest is set to the highest). For a short time, the worker, in a sense, exists in two places at once, and protects memory associated with both branches from reclamation by other workers.

This way of dealing with idle workers has two major advantages. Firstly, it makes the public region above the wavefront entirely read-only except for public backtracking (and synchronisation). Secondly, it neatly divides the wavefront into regions. An idle worker need only scan the wavefront to its left and right as far as the nearest idle worker in either direction.

A worker may suspend, in which case its sentry node is simply marked as suspended, and the worker proceeds to look for other work. The sentry node may be a wavefront node or a wavelet node, but in the latter case it will fairly quickly find itself in the wavefront, as its parent node becomes exhausted. A suspended node may exist in the wavefront for an arbitrary amount of time, but eventually either it will be cut or the suspension will be lifted and work resumed at the suspended node. Note that, except for suspensions, the number of nodes in the wave front is bounded by the total number of workers.

The implementation of cut and side effects depends on being able to determine whether or not a worker is leftmost within some scope (possibly global). This is achieved by sweeping the wavefront to the left. The join pointers will show when the scope boundary has been reached. Were it not for possible sequential choicepoints, idle workers could be ignored for leftmost determinations.

At the present time the Wavefront scheduler is still in an early stage of development. A good deal of refinement and experimentation remains to be done. To begin with there are a number of features of the Manchester scheduler (shadowing and lazy release) which could be incorporated. Overall, we hope that the Wavefront scheduler will provide for greater flexibility in experimentation with scheduling concepts. In particular, we are looking into more sophisticated ways of dealing with speculative work.

2.4.3 The Graphical Tracing Facility

Aurora also encompasses a set of tools for understanding the behavior of the system. They include a mechanism for recording events in the scheduler, and a graphical tracing facility for replaying those events on a Sun workstation to show pictorially how the workers explore the search tree [11].

In Figure 2.1 we show a typical snapshot of the Argonne scheduler at work, taken near the beginning of the search for all solutions of the "Zebra" puzzle by 16 workers. We are looking at the public part of the tree. Each "bee" at the end of branch represents a worker executing sequentially in the private section. The nodes shaped like honey pots (attractive to bees) have unexplored alternatives available, and nodes with parallel bars across them have alternatives that can be explored in parallel. The stalk of non-parallel nodes beginning at the root arises from the (sequential) Prolog shell.

From this snapshot we can tell that lots of work is currently available, and that the workers are distributed fairly evenly in the tree. We can see that worker 12 has just taken the last alternative from a node, and so is interrupting worker 7, telling it to make the node at the top of its stack public. Worker 13 is just finishing

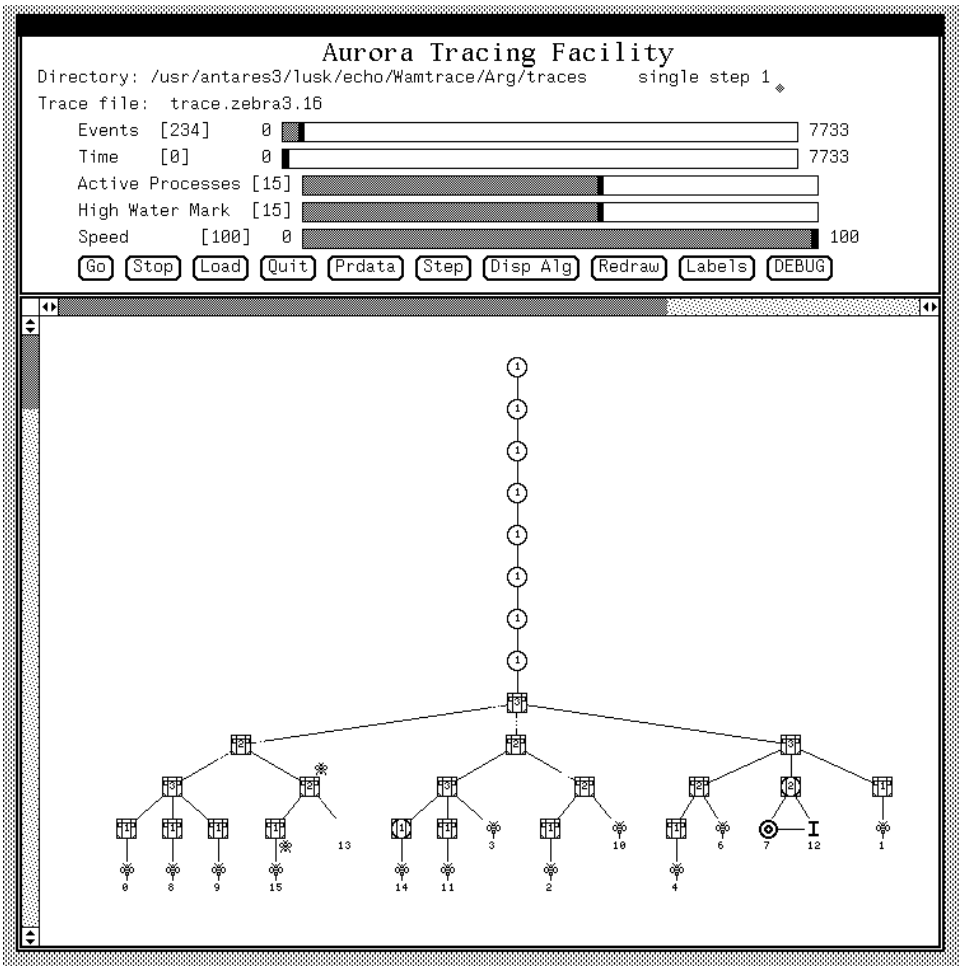

Figure 2.1: Snapshot of the graphical tracing facility

its branch, and on the node below there is another idle worker (actually, worker 5) that is about to take up work.

A much better feel for the computation is gained by watching the display in action and seeing the workers move about the tree in search of work. By clicking on the appropriate buttons, one can stop and restart the display, single step for close scrutiny of critical events, and display the predicate names associated with each choicepoint. These labels are important for relating the tree to the original program, but clutter the display and so are not shown here.

The graphical tracing facility has been very useful for investigating the behavior of the different schedulers. It has been particularly helpful in identifying "performance bugs", in which a computation is carried out correctly, but not as fast as it should be. In many cases the graphical display brings out the problem quite clearly.

2.5 Experimental Results

In Tables 2.1, 2.2 and 2.3, we present some performance data for Aurora running on a Sequent Symmetry under the three schedulers. The data is illustrative, and should not be regarded as providing a definitive comparison of the schedulers, or indeed a definitive picture of Aurora behaviour. The tables show times and speedups for different numbers of processors. For the Manchester scheduler two of the refinements described in Sec. 2.4.2.1 have been switched on for these runs: shadowing and lazy-release. The benchmarks considered are `8-queens2`, a naïve (generate and test) version of the 8 Queens problem from ECRC; `salt-must2`, a version of the Salt and Mustard puzzle from Argonne (adapted to remove meta-calls); `tina`, a holiday

planning program from ECRC; `db5`, the database query part of a Chat-80 natural language query[8]; `parse5`, the natural language parsing part of the same Chat-80 query.

Table 2.4 shows the relative speed of other Prolog systems compared with Aurora running on one Sequent Symmetry processor, for the same benchmarks. (The Aurora times are taken to be the average for the three schedulers, there being no significant difference between the schedulers on one processor). The main comparison is with the underlying sequential Prolog implementation, Sicstus 0.3, also running on a Sequent Symmetry. For additional comparison, we show the relative speed on a Sun 3/75 of Sictus 0.3, Quintus Prolog 2.4, and the most recent version of Sicstus, version 0.6.

In Table 2.5 we present some sample profiling data obtained from running the same benchmarks on an instrumented version [24] of Aurora (Manchester scheduler) with 20 processors on a Sequent Symmetry. First we give the execution time within the instrumented system (after substracting the measurement overheads), followed by some statistical data: the number of procedure calls (including built-ins); the number of tasks (engine invocations); the average number of calls per task (the quotient of the two previous quantities). The next three columns show the total overhead needed to support or-parallelism divided into three main categories: execution related overheads (i.e. the SRI binding scheme and the periodic test for scheduling activities during Prolog work), task switching overheads, and idle time. These columns show the total time (including locking and migration) spent in each main activity by all processors, expressed as a percentage of the sequential (Sicstus 0.3) execution time. The last two columns provide, as additional information, the total time spent respectively in locking and migration (i.e. the installation/deinstallation of bindings needed on task switching), again expressed as a percentage of sequential execution time.

The performance results that these tables illustrate are encouraging. On one processor, Aurora is only about 25% slower than Sicstus 0.3, the sequential system from which it is derived. Sicstus 0.3 is itself only about 2.7 times slower than Quintus Prolog, one of the fastest commercial systems. The latest version of Sicstus, Sicstus 0.6, is even faster, only about 1.8 times slower than Quintus, and we expect this improvement to carry over to Aurora when migration to version 0.6 has been completed.

On 20 processors, the Aurora speedup (relative to its speed on one processor) depends on the application, but can be over 18 on programs with almost ideal or-parallelism, while substantial speedups of 10 or more are obtained on a range of benchmarks including some drawn from real applications in natural language parsing and database query processing. These speedups represent a real performance improvement over sequential systems: for example, for the benchmarks shown, Aurora on 20 processors is 4 to 7 times faster than Quintus Prolog on a Sun 3/75. We shall see in the next section that very good speedups (and absolute performance) are also obtained on a variety of complete, large-scale applications.

The results demonstrate that the overheads introduced by adapting a high performance Prolog engine for the SRI model are low. The profiling data show that the cost of updating binding arrays on task switching, which was feared to be a major source of overhead, is quite small in practice. The relatively high migration cost for the `parse5` benchmark is caused by a rather peculiar shape of the search tree: two long branches with some work appearing on both of these from time to time.

Similarly, the time spent in locking is acceptably low, at least for the Manchester scheduler. It should be noted that the locking overhead in the Manchester scheduler is now low partly because the profiling data has been very helpful in locating and eliminating congestion in those parts of the algorithm where the competition for locks was high.

We believe that the most significant overhead is the high, relatively fixed, cost of task switching that prevails for all the schedulers in the current Aurora implementation. As can easily be calculated from data in Table 2.5, the cost of a single task switching operation is on average around 7 to 10 (sequential) Prolog procedure calls, and this figure seems to be relatively independent of the nature of the task switch. This places a limit on the granularity of parallelism that is worth exploiting, tasks of less than about 10 procedure calls being hardly worth exploiting. It will be seen that the `db5` and `parse5` are close to this limit. In addition to the direct cost of task switching, there seems to be a significant amount of idle time which cannot be explained by lack of parallelism, and which, we believe, is caused by delays in creation of work, due to earlier task switching overheads. Consequently the high cost of the task switching operation and the partitioning of work into sometimes unnecessarily small tasks are the main factors to blame for the less than perfect speedups. It is possible that task switching costs can be reduced by low-level tuning of the engine/scheduler interface amongst other things, irrespective of the high-level scheduling strategy.

[8] "Which European countries that contain a city the population of which is more than 1 million and that border a country in Asia containing a city the population of which is more than 3 million border a country in Western Europe containing a city the population of which is more than 1 million?"

	Processors				
Benchmark	1	4	8	16	20
8-queens2	29.18	7.31*(3.99)*	3.69*(7.91)*	1.95*(15.0)*	1.58*(18.5)*
sm2 *10	11.61	3.00*(3.87)*	1.59*(7.30)*	1.00*(11.6)*	0.80*(14.5)*
tina	20.91	5.56*(3.76)*	3.01*(6.95)*	1.78*(11.7)*	1.55*(13.5)*
db5 *10	3.78	1.07*(3.53)*	0.64*(5.90)*	0.44*(8.61)*	0.40*(9.47)*
parse5	5.88	1.64*(3.59)*	1.03*(5.70)*	0.75*(7.85)*	0.64*(9.19)*

Table 2.1: Times (in seconds) and speedups for Aurora, Manchester scheduler

	Processors				
Benchmark	1	4	8	16	20
8-queens2	29.11	7.37*(3.95)*	3.74*(7.78)*	1.96*(14.9)*	1.59*(18.3)*
sm2 *10	11.62	4.00*(2.91)*	3.14*(3.70)*	0.90*(12.9)*	0.75*(15.5)*
tina	21.08	5.51*(3.83)*	2.98*(7.07)*	1.76*(12.0)*	1.55*(13.6)*
db5 *10	3.85	1.06*(3.63)*	0.68*(5.65)*	0.45*(8.54)*	0.38*(10.1)*
parse5	5.89	1.82*(3.24)*	1.32*(4.45)*	1.07*(5.51)*	1.02*(5.75)*

Table 2.2: Times (in seconds) and speedup for Aurora, Argonne scheduler

	Processors				
Benchmark	1	4	8	16	20
8-queens2	29.12	7.32*(3.98)*	3.78*(7.70)*	2.08*(14.0)*	1.74*(16.8)*
sm2 *10	11.66	3.04*(3.84)*	1.52*(7.67)*	1.02*(11.4)*	1.04*(11.2)*
tina	21.13	5.44*(3.88)*	2.89*(7.30)*	1.72*(12.3)*	1.59*(13.3)*
db5 *10	3.67	0.98*(3.73)*	0.57*(6.49)*	0.40*(9.12)*	0.39*(9.35)*
parse5	6.01	1.65*(3.63)*	1.05*(5.71)*	0.79*(7.58)*	0.57*(10.5)*

Table 2.3: Times (in seconds) and speedup for Aurora, Wavefront scheduler

	TIME (sec)	RELATIVE SPEED			
	Aurora 0.0	Sicstus 0.3	Sicstus 0.3	Quintus 2.4	Sicstus 0.6
Benchmark	Symmetry	Symmetry	SUN 3/75		
8-queens2	29.14	1.25	0.91	2.68	1.45
sm2 *10	11.63	1.26	0.92	2.34	1.22
tina	21.04	1.26	0.84	2.29	1.39
db5 *10	3.77	1.17	0.90	2.42	1.28
parse5	5.93	1.23	0.99	2.62	1.52

Table 2.4: Comparing speed of other Prolog implementations with Aurora

	TOTAL TIME	TOTAL	TOTAL	CALLS	EXECUTION	TASK SWITCHING	IDLE	LOCKING	MIGRATION
Benchmark	(sec)	CALLS	TASKS	/TASK	OVERHEADS	OVERHEADS	TIME	TIME	TIME
					% of sequential execution time (Sicstus 0.3)				
8-queens2	1.580	167207	1822	92	25.12%	8.45%	1.68%	0.79%	0.49%
sm2 *10	0.763	135740	3440	39	22.29%	26.03%	17.30%	7.13%	1.71%
tina	1.591	160662	3349	48	31.96%	23.28%	34.57%	10.03%	1.53%
db5 *10	0.441	55450	3145	18	24.04%	71.20%	77.79%	38.55%	5.30%
parse5	0.654	39096	3384	12	31.64%	107.37%	31.65%	10.52%	31.75%

Table 2.5: Profile data for Aurora, Manchester scheduler, 20 processors

Remarkably similar speedups on the same benchmarks have been obtained for ECRC's PEPSys model [22]. The fact that two quite different models should produce similar speedups suggests that the speedups are limited mainly by the intrinsic granularity of the parallelism available in the examples. Simulation data by Kish Shen suggests that all the examples potentially have at least 20-fold parallelism, but that granularity varies widely and is very fine in the benchmarks with poorer speedups.

2.6 Applications

Besides running small benchmarks, we have ported a number of large-scale Prolog applications to Aurora to see how easy the porting is and to investigate how performance fares in real life.

Apart from the applications described in more detail below, other applictions tested are an Andorra Prolog interpreter [14], the Satchmo theorem prover (two versions: the first for theorems in Predicate Logic and the second for Propositional Logic), and a lexicon learning program. They show speedups from good (8 on 12 processors) to very good (11 on 12 processors).

2.6.1 The Pundit Natural Language System

The Pundit natural language system [19] developed by the Unisys Paoli Research Centre consists of a parser and a broad coverage restriction grammar. The grammar used consists of 125 BNF rules and 55 restrictions plus meta-rules handling conjunctions. There are about 350 disjunctions in the grammar. A kind of semantic parsing (selection) can be used to reduce the search space, but unfortunately it has not been possible to use this component in our experiments.

This large application is perfectly suited for or-parallel execution. The speedups are nearly ideal. The only change we had to do was replacing calls to the standard (synchronous) `recorded` with asynchronous versions. The results are actually better than the predictions based on the somewhat pessimistic model of parallelism proposed by the Paoli group [19].

For typical sentences, speedups with 12 processors are in the range 9 to 11. For example, a speedup of 11.15 is obtained for the sentence *"Loud noises were coming from the drive end during coast down"*.

2.6.2 The Piles Civil Engineering Application

The Piles program developed by the University of Bristol Civil Engineering Department analyses possible faults in concrete piles from acoustic data. The program is essentially an expert system consisting of a rulebase, a rule interpreter and a set of facts to be interpreted against the rules. A set of test data is available for 31 different piles. Seven main classes of fault can be analysed. In practice, the system is used to find all seven possible classes of fault in all the piles tested (31 in this case). This normal mode of use gives very good or-parallelism (albeit of a rather trivial kind).

The original Piles program was written in a different Prolog dialect (running on an IBM PC) and had to be converted for running under Sicstus. The use of the `clause` predicate had to be eliminated in order to facilitate compilation of the program. In order to exploit the indexing feature of Sicstus, argument positions in certain predicates were reordered; this actually yielded a very big improvement. In order to overcome the deficiency of the floating point operation of the current Aurora implementation, all floating point numbers were changed to integers. The core of the program made use of `assert` and `retract` to keep track of the certain maxima in the search space. Such use of side effect predicates hampered the exploitation of parallelism. The side effect predicates were initially replaced by a `free_bagof` (one which can collect its solutions asynchronously in an arbitrary order) and then by a new built-in predicate called `maxof`.

With all these changes, the time on one processor dropped dramatically from 173 sec. to 8.13 sec., and the speedup on 11 processors improved from around 1 with the original program to 9.5 with the eventually refined program, the time on 11 processors being reduced to 0.85 sec. Given that the original program on an IBM PC took on the order of 20 minutes, the total performance improvement achieved is very striking, a factor of around 1500.

The speedup described so far arises mainly from analysing 31 piles in parallel, which is how the application is actually used. In order to observe how much parallelism is available at the core of the program, we

investigated the performance on a single pile seeking a single type of fault. Used in this way the program still shows a reasonably good speedup of around 7 with 11 processors. The use of sequential declaration, in an attempt to focus the exploitation of parallelism, did not help in this application.

2.6.3 Study of the R-classes of a Large Semigroup

In the context of the overall automated reasoning research at Argonne, there has been interest in using various artificial intelligence based tools as aids in the understanding the structure of certain large finite semigroups that play a fundamental role in classifying varieties of semigroups [21]. A recent theorem-proving run yielded twenty megabytes of output about the semigroup F_3B_21. Although the number of elements (102,303) was new and important information, we wanted to extract information about the $\mathcal{R}$-classes in order to understand the structure. This required first the extraction of 2844 distinguished elements and for each of these, a specialised theorem-proving run to identify the graph structure of its $\mathcal{R}$-class. Since the theorem-proving runs required specialised inference rules and subsumption criteria, it was convenient to write this program in Prolog. Since the computation was so large, it was well worth speeding up, and since it consisted of 2844 independent and relatively large computations, there was ample exploitable parallelism. On 24 Sequent Symmetry processors, the speedup was 23.4, the time for the computation being reduced from nearly two hours to under five minutes. The computation time on the fastest sequential Prolog system we could find, Quintus Prolog on a Solbourne Sun-4 clone, was nearly twenty minutes. Thus Aurora was 3.7 times faster than the fastest available sequential Prolog system.

2.7 Conclusion

Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors. It currently runs on Sequent and Encore machines. It has been constructed by adapting Sicstus Prolog, a fast, portable, sequential Prolog system developed at the Swedish Institute of Computer Science. The techniques for constructing a portable multiprocessor version follow those pioneered by Argonne National Laboratory in a predecessor system, ANL-WAM. The SRI model, as developed and refined at Manchester University, was adopted as the means to generalise the Sicstus Prolog engine for or-parallel operation.

Aurora has demonstrated the basic feasibility of the SRI model. A high absolute speed per processor is attainable, and significant speedups can be obtained through parallelism on real examples. The overheads of updating binding arrays on task switching seem quite tolerable in practice.

Aurora supports the full Prolog language and is able to run large Prolog applications and systems. We have demonstrated that substantial or-parallelism is available in a variety of real applications, and that this leads to good speedups and absolute performance when running on Aurora.

As regards the ultimate goal of obtaining truly competitive bottom-line performance, Aurora on a 20-processor Sequent Symmetry is typically 4 to 7 times faster than Quintus Prolog on a Sun 3/75 for a wide range of examples with sufficient or-parallelism. On 4 processors Aurora easily outperforms Quintus on all examples with sufficient parallelism, while on one processor Aurora is only about 2.5 times slower. As a point of comparison, Quintus Prolog is one of the fastest commercial Prolog system, while the Sun 3/75 was until recently considered to be a fast processor. Turning to the fastest Prolog system we could find today, Quintus Prolog on a Solbourne Sun-4 clone, Aurora was 3.7 times faster on a large theorem proving application.

However, it can be argued that Aurora will not become truly competitive with sequential Prolog systems until shared-memory multiprocessors become cheaper and more cost-effective, bearing in mind that a 20-processor Sequent Symmetry is an order of magnitude more expensive than a fast workstation. The main factor preventing Aurora from being truly competitive is that multiprocessor machines, as an emerging technology, are still relatively expensive and have lagged behind in keeping pace with the dramatic yearly increase in sequential processor speeds. However this situation is changing. The next generation of fast processors is likely to appear simultaneously in workstations and multiprocessors that will support Aurora, and at the same time multiprocessors are likely to become increasingly competitive in terms of price/performance.

The other factor limiting Aurora competitiveness is the fact that (on equivalent hardware) the Aurora engine is some 3 times slower than the Quintus engine, due primarily to its being a portable implementation written in C, but reflecting also the overheads of the SRI model. We expect this factor to be reduced by the migration

to Sicstus version 0.6, and further improvements would be possible if the engine were implemented at as low a level as Quintus. Thus, with suitable tuning of the Aurora engine, truly competitive performance is likely to be obtainable on the next generation of multiprocessors.

The experience of implementing Aurora has demonstrated that it is relatively easy to adapt a state-of-the-art Prolog implementation, preserving the complete Prolog semantics. The main novel component is the scheduler code, which is responsible for coordinating the activities of multiple workers looking for work in a Prolog search tree. A clear and simple interface has been defined between the scheduler and the rest of the system. This makes it easy to experiment with alternative schedulers (three quite different schedulers currently exist), and should make it easier to apply the same parallelisation techniques to other existing Prolog systems.

Aurora is a prototype system, and there are many issues that need further exploration. In particular, more experimentation is needed with different scheduling strategies and mechanisms. It may be possible to reduce the high cost of task switching by more efficient implementation, or by alternative scheduling strategies which do not follow the "topmost node" heuristic. For example, current implementations of the BC model [1] achieve a much larger effective task size by dividing work at the "bottom-most node" rather than the topmost.

The current schedulers are able to handle cut, commit and side effects correctly. However, they require major enhancement to handle speculative work efficiently. The present schedulers treat all work as being equally worthwhile, and make no allowance for how speculative a piece of work may be. A more intelligent scheduling strategy should be prepared to suspend work that has become highly speculative if there is work available that is likely to be more profitable. Thus there is a need for "voluntary" suspension in addition to the present "compulsory" suspension. Possible scheduling schemes giving preference to non-speculative work or failing that to the least speculative work available are being discussed [15] and are going to be implemented and evaluated in the Aurora system.

The existing Aurora system allows researchers to experiment with or-parallel logic programs. We are making the system available to other research groups. We expect to continue to improve its capabilities and speed, and to port the system to new shared-memory multiprocessors as they become available.

The work done so far has inspired many directions for future research. One major extension that we are pursuing is the incorporation of and-parallelism, in the form of the Andorra model and language [14, 31]. The work has also inspired ideas for a novel architecture supporting shared virtual memory, called the data diffusion machine [29]. We believe Aurora can contribute generally to the study of parallelism in logic programming languages.

2.8 Acknowledgements

This work was greatly stimulated and influenced by many other colleagues involved in or associated with the Gigalips Project. We thank all of them.

This work was supported in part by the U.K. Science and Engineering Research Council, under grant GR/D97757, in part by ESPRIT project 2471 ("PEPMA"), and in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

References

[1] Khayri Ali. *Or-Parallel Execution of Prolog on BC-Machine.* SICS Research Report, Swedish Institute of Computer Science, 1987.

[2] 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.

[3] Alan Calderwood. Aurora—description of scheduler interfaces. January 1988. Internal Report, Gigalips Project.

[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. Internals of Sicstus Prolog version 0.6. November 1987. Internal Report, Gigalips Project.

[6] Mats Carlsson and Johan Widén. SICStus Prolog User's Manual. October 1988. SICS Research Report R88007B.

[7] Andrzej Ciepielewski and Seif Haridi. A formal model for or-parallel execution of logic programs. In *IFIP 83 Conference*, pages 299–305, North Holland, 1983.

[8] Andrzej Ciepielewski, Seif Haridi, and Bogumil Hausman. Initial evaluation of a virtual machine for or-parallel execution of logic programs. In *IFIP-TC10 Working Conference on Fifth Generation Computer Architecture*, Manchester, U.K., 1985.

[9] William Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.

[10] Doug DeGroot. Restricted and-parallelism. In Hideo Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478, Institute for New Generation Computing, Tokyo, 1984.

[11] Terrence Disz and Ewing Lusk. A graphical tool for observing the behavior of parallel logic programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 46–53, 1987.

[12] Terrence Disz, Ewing Lusk, and Ross Overbeek. Experiments with OR-parallel logic programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 576–600, MIT Press, 1987.

[13] Steven Gregory. *Parallel Logic Programming in Parlog*. Addison-Wesley, 1987.

[14] Seif Haridi and Per Brand. Andorra Prolog–an integration of Prolog and committed choice languages. In *International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[15] Bogumil Hausman. Pruning and scheduling speculative work in or-parallel Prolog. In *PARLE 89, Conference on Parallel Architectures and Languages Europe*, Springer-Verlag, 1989.

[16] Bogumil Hausman, Andrzej Ciepielewski, and Alan Calderwood. Cut and side-effects in or-parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[17] Bogumil Hausman, Andrzej Ciepielewski, and Seif Haridi. Or-parallel Prolog made efficient on shared memory multiprocessors. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 69–79, 1987.

[18] Manuel Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 25–39, Springer-Verlag, 1986.

[19] Lynette Hirschman, William Hopkins, and Robert Smith. Or-parallel speed-up in natural language processing: a case study. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 263–279, MIT Press, August 1988.

[20] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David 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.

[21] Ewing Lusk and Robert McFadden. Using automated reasoning tools: a study of the semigroup F2B2. *Semigroup Forum*, 36(1):75–88, 1987.

[22] Michael Ratcliffe. A progress report on PEPSys. July 1988. Presentation at the Gigalips Workshop, Manchester.

[23] Ehud Shapiro, editor. *Concurrent Prolog–Collected Papers*. MIT Press, 1987.

[24] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732, MIT Press, October 1989.

[25] David H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.

[26] David H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool.* PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.

[27] David H. D. Warren. Or-parallel execution models of Prolog. In *TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, Pisa, Italy*, pages 243–259, Springer-Verlag, March 1987.

[28] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[29] David H. D. Warren and Seif Haridi. Data Diffusion Machine–a scalable shared virtual memory multiprocessor. In *International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[30] Harald Westphal, Philippe Robert, Jacques Chassin, and Jean-Claude Syre. The PEPSys model: combining backtracking, and- and or-parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*, IEEE, 1987.

[31] Rong Yang. Solving simple substitution ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 113–128, MIT Press, June 1989.

Chapter 3

Performance Analysis of the Aurora Or-Parallel Prolog System[1]

Péter Szeredi[2]

Department of Computer Science
University of Bristol, Bristol BS8 1TR, U.K.

Abstract

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors based on the SRI model of execution. The capabilities of Aurora in exploiting parallelism vary from application to application: some show almost linear speed-up, whilst for others the speed-up is much worse than the theoretical maximum.

The Manchester version of the Aurora system has been instrumented to provide various types of profiling information. Main sources of overhead in parallel execution have been identified and the frequency of specific events, as well as the time spent in each of them has been measured. Special attention has been paid to the binding array update overheads associated with the SRI model and to the overheads of synchronisation using locks.

We present a short description of the instrumented Aurora system and evaluate the basic set of profiling data. Our main conclusion is that the high cost of task switching in the present implementation is the main cause of poor speed-ups. The cost of updating the binding arrays, which was feared to be the major cause of overhead, seems to be rather small. Similarly, locking costs are acceptably low and there is no significant increase in the average locking time.

3.1 Introduction

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, currently running on Sequent and Encore machines. It has been developed in the framework of the Gigalips project [5], a collaborative effort between Argonne National Laboratory in Illinois, University of Bristol (previously University of Manchester) and the Swedish Institute of Computer Science (SICS) in Stockholm.

The SRI model [11] has been adopted as the basis of Aurora. According to this model the system consists of several *workers* (processes) exploring the search tree of a Prolog program in parallel. Each node of the tree corresponds to a Prolog choicepoint with a branch associated with each alternative clause. As the tree is being explored, each node can be either *live*, i.e. having at least one unexplored alternative, or *dead*. Live

[1] This paper has appeared in the proceedings of NACLP'89 [10]
[2] *On leave from* SZKI, Donáti u. 35-45, Budapest, Hungary

nodes correspond to pieces of work a worker can select. Each worker has to perform activities of two basic types:

- executing the actual Prolog code,
- finding work in the tree, providing other workers with work and synchronising with other workers.

The above two kinds of activities have been separated in Aurora: those parts of a worker that work on executing the Prolog code are called the *engine*, whilst those concerned with the parallel aspects are called the *scheduler*. The engine used in Aurora is a modified version of SICStus Prolog (version 0.3).

In accordance with the SRI model each worker has a separate *binding array* to store its own bindings to potentially shared variables (conditional bindings). This technique allows constant time access to a value of a shared variable, but imposes an overhead of updating the binding arrays whenever a worker has to move within the search tree. The number of bindings made on the given path of movement is called the *migration cost*, since it is proportional to the updating overhead of binding arrays.

The or-tree is divided to an upper, *public*, part accessible to all workers and a lower, *private*, part. A worker exploring its private region does not have to be concerned with synchronisation or maintaining scheduling data – it can work very much like a standard Prolog engine. The boundary between the public and private regions changes dynamically. One of the critical aspects of the scheduling algorithm is to decide when to make a node public, allowing other workers to share work at it. The current Aurora schedulers use a *dispatching on topmost* strategy: a node is made public when all nodes above it are dead, i.e. have no more alternatives to explore. This means that each worker tries to keep a single piece of work on its branch available to other workers.

Three separate schedulers are being developed currently for Aurora. The Argonne scheduler [3] relies on data stored in the tree itself to implement a local strategy according to which live nodes "attract" workers without work. When several workers are idle they will compete to get to a given piece of work and the fastest one will win. In contrast with this the Manchester scheduler [4] tries to select the nearest worker in advance, without moving over the tree. It uses global data structures to store information on available work and workers as well as data stored in the tree itself. The wavefront scheduler [2] tries to achieve the goal of optimal matching between work and workers using a special distributed data structure, the *wavefront*, which links all the live nodes and idle workers into a doubly linked list.

Our performance analysis work aims at understanding the factors influencing the behaviour of Aurora schedulers. The measurements were performed on the Manchester scheduler. Considerable part of the analysis, however, applies to design decisions that are common to all three schedulers. The major goals of our work are the following:

- to measure the costs associated with the SRI binding scheme, both during Prolog execution in the engine and within the scheduler activities,
- to evaluate the efficiency implications of some design issues in Aurora, such as the private-public division of the search tree and the engine-scheduler interface,
- to identify the major algorithmic components of the Manchester scheduler and assess their effect on the performance of the system,
- to analyse the locking overheads in the Manchester scheduler,

Section 3.2 outlines the structure of the Manchester scheduler and Section 3.3 briefly describes our instrumentation of the system. Section 3.4 introduces the benchmarks used in the analysis and presents some basic timing data, while Section 3.5 contains detailed data and evaluation of the basic overhead activities. Section 3.6 covers locking and migration overheads. In Section 3.7 some extensions to the basic scheduling algorithm are briefly described and evaluated. Finally Section 3.8 summarises the conclusions of the paper.

Report [9] is an extended version of this paper, giving more detailed evaluation of performance issues and data for individual benchmarks as well.

3.2 The working cycle of Aurora

The basic working cycle of the Aurora system (with the Manchester scheduler) is presented in Fig. 3.1. Each box in this figure represents a basic activity that has been measured in our profiling. Tasks concerned with the execution of side effect predicates have not been shown on the figure in order to simplify the presentation.

The main data structures used by the Manchester scheduler are introduced briefly to make the discussion self contained. Every node of the search tree has a number of scheduler specific fields of which the worker *bit-map* is of major importance. Each bit in this field corresponds to a specific worker and indicates whether the given worker is at or below the node. There are two global arrays both indexed on a unique worker identifier. The *worker* array stores information on workers including an indication of the migration cost (between the worker's current node and the root) if the worker is idle and an interrupt message area which allows one worker to notify another about a specific event. The other main global data structure is the array of *queues* each of which stores a pointer to a single live node (if any) the given worker has to share on its branch, together with the migration cost.

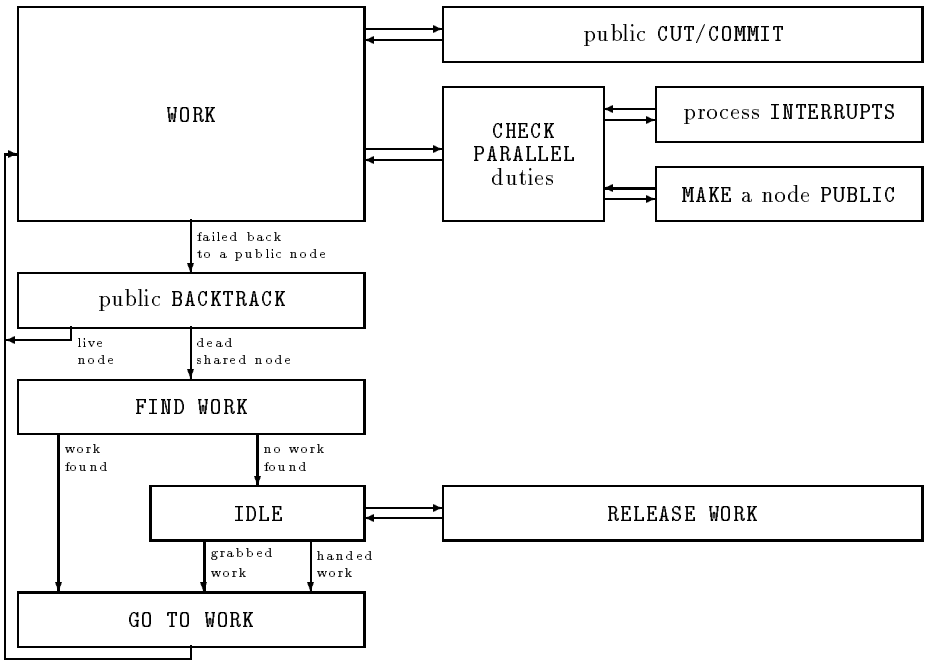

Figure 3.1: The working cycle of Aurora (Manchester scheduler)

The basic activity of each worker is to do the "real" *work*, i.e. Prolog execution (resolution and backtracking). Some of the built-in predicates to be executed need additional care due to the parallel environment, most notably *cut* and *commit* to a public node. Another kind of overhead arises from the fact that the worker has to *check* periodically if it has any duties concerning *parallel* execution (this is done at every nth Prolog call, $n = 5$ by default[3]). The duties include checking if it has received any *interrupts* (e.g. about having been cut), and also checking if it is feasible to *make* some of its newly created nodes *public* (i.e. available to other workers). When a node is made public, an idle worker is interrupted to do the releasing of work (see below). If there are no idle workers the address of the new public node is deposited in the worker's queue.

A continuous piece of work executed by a worker is called a *task*. A worker normally finishes its task when it fails back to a public node and then it does *public backtracking*. If the node in question is still live, the worker can claim the next alternative and return to work. If the node is dead and not shared with others any more, then the worker can recover the node and continue the public backtracking with the parent of the node. If, however, the node in question is dead and shared, then the worker has to abandon its current branch and try to *find work* in some other part of the search tree.

The worker uses the bit-map in its current node to determine which workers are below him and examines the corresponding queues to locate the nearest piece of work. If no work is found in this subtree, it repeats the same procedure for the parent of the current node. If work is found eventually, the worker grabs it (i.e.

[3] This frequency of checks has been found the most beneficial for the Manchester scheduler [8]

claims it for itself) and *goes to work*. This involves modifying its binding array as well as updating scheduler information in the affected nodes of the tree. When the worker is positioned at the appropriate node, it enters its main working mode again.

When no work is available some workers may become *idle*. When, later on, a node is made public by an active (working) worker, the idle worker who is nearest to this node needs to be selected and handed the work. To let the active worker continue with its Prolog work as soon as possible, this selection process, called *releasing work*, is actually done by one of the idle workers. The releasing worker will consider progressively larger subtrees around the new public node in its search for the nearest idle worker (including itself). The worker array and the bit-maps will be used during this search much in the same way as for finding work. When the appropriate idle worker is selected, it is handed the given piece of work. This idle worker will then leave its idle state, go to the appropriate node of the tree and start working. Occasionally an idle worker may notice a piece of work appearing in a queue and grab it on its own initiative, leaving idle state and going to work.

3.3 Instrumenting Aurora

The Aurora implementation has been instrumented to collect various performance data. The execution cycle of the system has been divided to a number of disjoint activities as depicted in Fig. 3.1 (`WORK`, `CUT/COMMIT`, `CHECK PARALLEL`, etc.). The number of times a given type of activity has been performed as well as the total time spent in it have been collected. Furthermore two sub-categories have been distinguished: separate time accumulators and event counters have been provided for locking and for moving (updating the binding arrays) within each of the main categories.

Additional counters have been inserted for the basic Prolog events: `calls` (Prolog procedure calls – including most of built-in predicates), `backtracks`, creation of `nodes` (choicepoints), `conditional bindings` (i.e. bindings to potentially shared variables) and also for `move bindings` i.e. those installed by workers moving around the tree.

The profiling experiments were run under the DYNIX^{TM} operating system on a Sequent Symmetry^{TM} S27 multiprocessor equipped with twelve processors and 16 Mbytes of memory. Since we wanted each worker to be assigned the full power of a CPU, we used at most 11 workers to leave one processor aside for the computing requirements of the operating system.

Efforts have been made to account for the measurement overheads by excluding the time actually spent in measurement from the times accumulated for specific activities. The net running times obtained by subtracting the average overhead from the times measured are only a few percent higher than the times obtained from an uninstrumented system. We believe that this distortion resulting from the instrumentation is reasonably low and thus measurements do reflect what is happening in the original, uninstrumented system.

3.4 The benchmarks

We used the Chat-80 natural language query system as one of the main sources for the performance analysis of Aurora. Benchmarks `parse1` – `parse5` run the parsing component of Chat-80 to find all possible parses of various queries. The Prolog translations of two queries were taken as representatives of database search type applications (`db4` and `db5`). The translation used was that produced by Chat, but with the extra level of query interpretation and all cuts removed.

Further benchmarks were adapted from those used by the PEPSys group at ECRC: `farmer`, a small program for planning the farmer's crossing a river with a wolf, a goat and a cabbage; `house`, the "who owns the zebra" puzzle, using constraint directed search; `8-queens2`, the naive (generate and test) version of 8 queens; `8-queens1`, a more efficient algorithm for 8 queens, which checks the state of the chess-board in each step; `tina`, a holiday planning program. Finally `sm2`, the salt-mustard puzzle originating from Argonne, has also been included. It has been slightly modified to avoid the frequent meta-calls of the original version.

All the above benchmarks look for all solutions of the problem. Although there are cuts in some of the programs, these have small scope and so the amount of additional (speculative) work done when running in parallel is minimal (cf. Section 3.5, Table 3.3).

Table 3.1 shows the running times in seconds, with speed-ups (relative to the 1 worker case) given in

parentheses. The last column shows the running time on SICStus Prolog version 0.3, the Prolog system on which the engine of Aurora is based. The "speed-up" figure in this column, (the ratio of the running time on Aurora with 1 worker and the running time on SICStus) is actually indicating the overheads of extending the Prolog system to allow or-parallel execution.

		Aurora				
	Goals	Workers				
Group	* repetitions	1	4	8	11	Sicstus 0.3
H	8-queens1	11.33	2.87*(3.95)*	1.47*(7.71)*	1.11*(10.2)*	9.08*(1.25)*
	8-queens2	33.60	8.39*(4.00)*	4.27*(7.87)*	3.19*(10.5)*	26.04*(1.29)*
	tina	23.73	6.16*(3.85)*	3.29*(7.21)*	2.53*(9.38)*	18.89*(1.26)*
	sm2 *10	13.57	3.52*(3.86)*	1.86*(7.30)*	1.43*(9.51)*	10.49*(1.29)*
	AVERAGE		*(3.92)*	*(7.52)*	*(9.91)*	*(1.27)*
M	parse2 *20	10.14	3.07*(3.30)*	2.17*(4.68)*	1.98*(5.13)*	7.99*(1.27)*
	parse4 *5	9.38	2.64*(3.55)*	1.66*(5.65)*	1.44*(6.51)*	7.38*(1.27)*
	parse5	6.63	1.81*(3.66)*	1.13*(5.87)*	0.95*(6.99)*	5.25*(1.26)*
	db4 *10	3.62	1.01*(3.58)*	0.59*(6.14)*	0.47*(7.70)*	3.04*(1.19)*
	db5 *10	4.41	1.23*(3.59)*	0.71*(6.21)*	0.57*(7.74)*	3.69*(1.20)*
	house *20	9.22	2.61*(3.53)*	1.58*(5.84)*	1.31*(7.06)*	7.37*(1.25)*
	AVERAGE		*(3.54)*	*(5.73)*	*(6.86)*	*(1.24)*
L	parse1 *20	2.72	0.98*(2.78)*	0.86*(3.17)*	0.87*(3.14)*	2.13*(1.27)*
	parse3 *20	2.33	0.92*(2.53)*	0.82*(2.85)*	0.80*(2.93)*	1.81*(1.28)*
	farmer *100	5.37	2.63*(2.04)*	2.52*(2.13)*	2.53*(2.12)*	4.09*(1.31)*
	AVERAGE		*(2.45)*	*(2.72)*	*(2.73)*	*(1.29)*

Table 3.1: Run times for benchmarks

For simpler benchmarks the timings shown refer to repeated runs, the repetition factor being shown in the first column. Additionally, each timing was carried out several times and the shortest of these is displayed in the table.

The benchmarks in Table 3.1 are divided into three groups according to the speed-ups shown:

- `8-queens1`, `8-queens2`, `tina` and `salt-mustard` show very good speed-up (around 10 for 11 workers);
- `parse2`, `parse4`, `parse5`, `db4`, `db5` and `house` show relatively good speed-ups (5–8 for 11 workers);
- `parse1`, `parse3` and `farmer` show rather bad speed-ups (2–3 for 11 workers).

Because of space limitations the average data for these groups will be presented in the following sections, rather then to show the benchmarks individually. The three groups will be referred to as Group H (high speed-up), Group M (medium speed-up) and Group L (low speed-up).

Some of the benchmarks used in this analysis were evaluated by Kish Shen [7] using his simulator similar to that described in [6]. Table 3.2 shows the results of simulation in two variants. The first part of the table assumes no overheads associated with task switching, while the second part assumes an overhead of 8 resolution time units. The table shows simulation results for 4–11 workers and also gives the maximum achievable speed-up with the actual number of workers needed to produce the speed-up shown in parentheses.

The simulation results confirm our grouping: the benchmark in Group H shows a very high level of parallelism (over 500-fold under ideal conditions), the benchmarks in Group M have a medium amount of parallelism, while the ones in Group L have a rather low level of parallelism (the ideal maximum speed-up being below 12). The discrepancy between the actual results and the figures predicted by the simulator for the 8 resolution units overhead is the smallest for the 4 workers case: a few percent for Groups H and M and 10-26% for Group L. This gap widens as more workers are considered, reaching 30% for Group M and 50% for Group L. We will return to comment on this difference after we have presented the actual overheads.

Group	Goals	No overheads Workers 4	8	11	Max speed-up		Overhead = 8 resolutions / task Workers 4	8	11	Max speed-up	
H	8-queens1	4.00	7.98	10.96	>536	(w>800)	3.99	7.95	10.90	>440	(w>800)
M	parse2	4.00	7.93	10.67	26.38	(w=66)	3.45	5.97	7.17	13.33	(w=126)
	parse4	4.00	7.98	10.89	41.26	(w=216)	3.84	6.63	8.47	22.68	(w=285)
	parse5	4.00	7.99	10.98	58.46	(w=256)	3.85	6.96	8.98	32.40	(w>100)
	db4	3.99	7.94	10.88	159.7	(w=800)	3.84	7.43	10.13	78.15	(w=400)
	house	3.96	7.76	10.51	53.72	(w=190)	3.69	6.73	8.71	30.54	(w=187)
	AVERAGE	3.99	7.92	10.79			3.73	6.74	8.69		
L	parse1	3.96	6.80	8.37	11.46	(w=37)	3.05	4.30	4.66	5.49	(w=50)
	parse3	3.92	6.97	8.71	11.46	(w=35)	2.86	4.07	4.44	5.34	(w=48)
	farmer	3.34	4.31	4.53	4.53	(w=10)	2.58	3.02	3.15	3.31	(w=16)
	AVERAGE	3.74	6.03	7.20			2.83	3.80	4.08		

Table 3.2: Speed-up of OR-parallel Prolog execution - simulation results

3.5 Basic overheads of or-parallel execution

Bench-mark Groups	CALLS per TASK	BACK-TRACKS per TASK	NODES per TASK	% of PUBLIC NODES	COND. BINDs per CALL	CALLS INCREASE	COND. BINDINGS INCREASE
Group H							
w = 1	30175.82	27867.74	23552.59	0.03%	0.39		
w = 4	547.84	485.65	407.68	0.51%	0.39	0.00%	0.49%
w = 8	202.11	163.20	149.67	1.24%	0.39	0.03%	1.56%
w = 11	100.82	75.10	70.61	2.01%	0.40	0.07%	2.98%
Group M							
w = 1	2429.24	1293.92	1017.37	0.21%	0.57		
w = 4	44.16	24.65	18.47	3.85%	0.60		6.01%
w = 8	19.49	10.84	7.96	8.72%	0.65		14.18%
w = 11	15.52	8.62	6.31	10.90%	0.67		17.81%
Group L							
w = 1	227.83	142.42	135.42	0.36%	0.73		
w = 4	9.62	5.69	5.11	12.64%	0.91	0.45%	29.36%
w = 8	6.82	4.03	3.62	18.80%	0.98	0.76%	45.28%
w = 11	6.49	3.85	3.48	20.18%	1.00	0.98%	47.59%

Table 3.3: Basic statistical data

Table 3.3 presents a set of engine related frequency data to help in understanding the timing data in the sequel. The first three columns of the table contain the average number of calls, nodes and backtracks per task and so provide a good indication of average task size. The next column gives a related piece of data: the percentage of public nodes among all nodes, i.e. the proportion of the public part within the search tree. The figures in these columns indicate that there is a dramatic decrease in task size as the number of workers increases and the three groups are visibly separated. It is worth noting, however, that the decrease for Group L is much less sharp than for the other two groups. This is because the average task size in Group L is not much more than 5 Prolog calls which is the default period between invocations of `CHECK PARALLEL`. This means that there will be no parallel activity (and so no node will be made public) in the first 5 calls within each task.

The fifth column contains the number of conditional bindings per Prolog call. Although the group averages do increase going from Group H to L, it is worth noting that while the "worst" benchmark, farmer, has 0.49 bindings per call, one of the "best", tina, has 0.67 (for 11 workers). The figures for the individual benchmarks vary from 0.18 (`db4` and `db5`) to 1.25 (`parse3`). While there is a 7-fold difference in the number of conditional bindings for these benchmarks, the variation in the actual overhead is much smaller: 19% to 28% (as shown in the last column of Table 3.1). This indicates that the significance of the actual binding overhead is much smaller than that of the general overhead of handling the binding array references during the unification process.

The CALLS INCREASE column shows the increase in the number of Prolog calls executed during the benchmark. This is an indication of how much unnecessary *speculative work* is performed by the system. Such work is undertaken when a worker choses a branch that later will be cut, thus causing code to be executed that would not be run in the sequential case. The amount of unnecessary speculative work is rather small for our examples, basically because they were chosen not to contain major cuts (e.g. there are none in Group M, at all). Nevertheless it is important to know for the evaluation of timing results that there is no significant increase in the actual Prolog work to be performed in our benchmarks when going from 1 to more workers.

The increase in the number of conditional bindings is due to the fact that when the last alternative is taken from a shared choicepoint, the latter can not be discarded (as it would be in the sequential case). This causes some additional bindings to become conditional. The original SRI-model envisaged that when all but one worker backtracked to a choicepoint these bindings could be *promoted* to become unconditional, but this has not yet been implemented in Aurora. The increase in the number of conditional bindings is naturally related to the number of public nodes (column 4) and so it is much bigger for the low granularity benchmarks than for the high granularity ones.

Bench-mark Groups	WORK	CHECK PAR.	MAKE PUBLIC	BACK-TRACK	FIND RELEASE WORK	GO TO	OTHER	IDLE	TOTAL
	% of sequential running time (Sicstus 0.3)								
Group H									
w = 1	129.74	1.03	0.02	0.06			0.03		130.88
w = 4	129.05	1.06	0.40	0.77	0.26	0.05	0.20	0.43	1.15	133.37
w = 8	129.72	1.07	0.97	1.61	0.98	0.20	0.78	0.96	3.82	140.11
w = 11	130.59	1.10	1.44	2.33	1.69	0.32	1.36	1.44	6.18	146.45
Group M									
w = 1	125.42	1.25	0.12	0.34			0.15		127.26
w = 4	126.67	1.28	2.06	4.18	1.87	0.51	1.97	2.12	3.48	144.15
w = 8	128.09	1.39	4.78	9.83	6.92	2.08	10.41	4.86	14.83	183.20
w = 11	129.38	1.46	6.08	13.21	9.85	3.39	17.07	6.15	25.14	211.72
Group L									
w = 1	130.32	1.06	0.26	0.72			0.33		132.70
w = 4	138.36	1.42	7.93	13.30	7.93	4.82	10.01	11.00	33.10	227.86
w = 8	146.14	1.82	12.59	20.05	15.28	17.94	28.71	20.45	162.76	425.74
w = 11	146.81	1.92	13.27	21.56	13.74	27.77	33.90	22.54	306.31	587.82

Table 3.4: Basic profile of the benchmarks

Table 3.4 shows how the time spent in solving a particular benchmark is divided between various activities of Fig.3.1. The time given for a specific activity is the total time for all workers including the time needed for locking and moving over the tree. Since there are only a few cuts and commits to public nodes in the benchmarks, figures for these and for processing interrupts have been included in the `OTHER` column which also covers the cost of the actual tests and procedure calls needed to implement the basic loop of the scheduler. To make the times for different benchmarks comparable they are expressed as a percentage of sequential execution time (using SICStus Prolog 0.3), i.e. what could be considered "real" Prolog work.

A few notes on some columns of Table 3.4 follow.

`WORK` – The overhead (i.e the percentage over 100%) appearing in the 1 worker case is basically the cost

of the SRI binding scheme. This is roughly proportional to the sequential execution time varying between 25% and 30% of the latter. The increase for the case of more workers is due to several factors: increase in the number of conditional bindings, speculative work, and decrease in granularity (causing some hardware related execution overheads, e.g. paging or the number of cache misses, to increase).

CHECK PARALLEL – The time spent in checking if any parallel activities need to be done during work is acceptably small, around 1-2%. The actual time of a single **CHECK PARALLEL** event is fairly constant for the case of 1 worker (5-6 μsec), the marked difference between Group M and the other two groups is caused by differences in average Prolog call times (since this activity is performed periodically, every nth procedure call). As the number of workers increases the tests involved in checking parallel duties become more complex and the average event time increases up to 10 μsec. This explains why Group M, having the highest cost for the 1 worker case, is "overtaken" by Group H.

MAKE PUBLIC ... **IDLE** – These columns of the table refer to activities related to task switching, i.e. to those along the main loop of Fig. 3.1: backtracking, looking for work, returning to work. (Although making a node public is not itself part of the main loop, it is a prerequisite for any tasks to be created at the given node.) The table shows these overheads to increase considerably when the granularity decreases, which is quite natural considering that lower granularity means more frequent execution of the main loop. The rise is most sharp for the **IDLE** column. This column, in fact, differs from the previous ones in that the time spent being idle depends on the amount of parallelism available in the program. Benchmarks in Group L are characterised by a rather low level of parallelism (cf. Table 3.2) which causes the sharp increase in the **IDLE** column.

Bench-mark Groups	AVERAGE TIME per TASK (msec)									PROLOG CALL (msec)
		TASK SWITCHING OVERHEADS							IDLE	
	EXEC-UTION	MAKE PUBLIC	BACK-TRACK	FIND WORK	REL. WORK	GO TO WORK	OTHER	TOTAL	OVER-HEAD	
Group H										
w = 1	5863.03	0.12	0.22				0.10	0.44		0.16
w = 4	112.92	0.12	0.18	0.10	0.03	0.07	0.11	0.61	0.31	0.16
w = 8	36.26	0.14	0.20	0.15	0.03	0.13	0.13	0.77	0.46	0.16
w = 11	17.73	0.15	0.20	0.18	0.03	0.15	0.13	0.84	0.52	0.16
Group M										
w = 1	418.04	0.07	0.20				0.09	0.37		0.14
w = 4	6.70	0.10	0.18	0.09	0.03	0.10	0.09	0.59	0.17	0.15
w = 8	2.77	0.10	0.20	0.15	0.04	0.22	0.10	0.80	0.31	0.15
w = 11	2.26	0.10	0.22	0.17	0.05	0.29	0.10	0.93	0.41	0.15
Group L										
w = 1	39.57	0.09	0.21				0.09	0.39		0.17
w = 4	1.86	0.10	0.18	0.10	0.06	0.13	0.14	0.71	0.42	0.18
w = 8	1.35	0.11	0.19	0.14	0.16	0.26	0.18	1.04	1.45	0.19
w = 11	1.29	0.11	0.19	0.12	0.24	0.29	0.19	1.15	2.64	0.19

Table 3.5: Parallel overheads per task

To get a clearer picture of the cost of various activities, average overheads per task have been calculated and shown in Table 3.5. Each column, except the last, shows how much time is spent in a specific activity during an average task (more exactly during one execution of the main loop of Fig. 3.1). The **EXECUTION** column shows the sum of the **WORK** and **CHECK PARALLEL** times. By comparing figures for various overheads to the ones in this column one can judge the impact of the given overhead on the total run time. The following columns show specific overheads – first the group of task switching overheads, and then the **IDLE** time. The last column gives the execution time of a Prolog procedure call to help interpret all the other times in the table.

The task switching overheads increase in various degree when the number of workers goes up. The cost of backtracking is quite stable, around 0.2 msec, and there is only a small increase in the cost of making nodes public. Finding work takes about 80% more time for 11 workers than for four in Groups H and M, but in

Group L this overhead shows a decrease for 11 workers. This effect is due to the fact that as the amount of parallelism runs out, finding work succeeds much less often - and an unsuccessful attempt to find work is much cheaper than a successful one. This also explains why the cost of releasing work is much higher for Group L than for the other two groups: as there is an abundance of idle workers for benchmarks in Group L, much larger proportion of tasks is created by releasing work rather than by finding work. The cost of going to work increases sharply for 4-11 workers, especially for Group M where it almost trebles. The total of task switching overheads increases by 30-60% between 4 and 11 workers – the major contributors to the increase being: `GO TO WORK`, `FIND WORK` (Group H and L) and `RELEASE WORK` (Group L).

Looking at the `IDLE` column in Table 3.5 it is quite interesting to note that all figures except the last two show the average idle time being between $\frac{1}{3}$ and $\frac{2}{3}$ of total task switching overheads. Let us examine to what extent this idle time is justified by the lack of parallelism in our benchmarks. The simulation results (Table 3.2) provide us with data on how much time would be spent idle under ideal conditions (no task switching overheads) – let us call this *primary* idle time. Denoting the full (ideal) run time for N workers by T_N, the primary idle time by P_N and the Prolog execution time by E, the following equations hold:

$$T_N = \frac{E + P_N}{N}$$

$$Speedup_N = \frac{T_1}{T_N} = \frac{N * E}{E + P_N}$$

since $P_1 = 0$, and thus

$$\frac{P_N}{E} = \frac{N}{Speedup_N} - 1 \tag{3.1}$$

Group	Goals	EXECUTION TIME per TASK (msec)	PRIMARY IDLE TIME %	TOTAL IDLE	PRIMARY IDLE TIME	NON PRIMARY IDLE TIME	TOTAL TASK SWITCHING OVERHEAD	NON PRIMARY IDLE TIME as % of TASK SWITCHING OVERHEAD
		AVERAGE TIME per TASK (msec)						
H	8-queens1	19.71	0.36%	0.38	0.07	0.30	1.03	29.55%
M	parse2 *20	1.66	3.09%	0.61	0.05	0.55	1.18	46.94%
	parse4 *5	2.52	1.01%	0.32	0.03	0.29	1.11	26.15%
	parse5	2.98	0.18%	0.20	0.01	0.20	1.29	15.08%
	db4 *10	1.64	1.10%	0.25	0.02	0.23	0.48	47.76%
	house *20	2.96	4.66%	0.74	0.14	0.60	0.99	60.91%
L	parse1 *20	1.28	31.42%	1.86	0.40	1.45	1.03	141.00%
	parse3 *20	1.33	26.29%	2.35	0.35	2.00	1.07	186.30%
	farmer *100	1.28	142.83%	3.71	1.83	1.87	1.34	139.52%

Table 3.6: PRIMARY IDLE TIME

This means one can calculate the amount of primary idle time with respect to the execution time if the ideal speed-up is given. When the above formula (3.1) is applied to the benchmarks in Table 3.2, it turns out that the calculated primary idle time accounts for only 5-20% of measured idle time (except for `farmer` where it is about 50%). The unaccounted part varies from 0.2 msec to 0.6 msec per task within Groups H and M, but is 1.5-2 msec per task in Group L (for 11 workers). One could distinguish between several reasons for this additional amount of idle time:

a. task switching overheads – because there is a delay in starting a task due to task switching overheads, there will be a delay in creating new pieces of work within that task, which may cause other workers to become idle. One would expect this kind of idle overhead to be proportional to the task switching overheads.

b. too fine grained parallelism – the scheduler will not exploit any parallelism during the first few (in our case 5) calls of a task. In fact this is a desired feature as long as task switching overheads take the equivalent of 5-10 procedure calls.

c. administration costs – time needed for entering and exiting the idle state may prolong the time spent idle, if that would otherwise be shorter than the time needed for administration.

In Groups H and M the non-primary idle time per task lies between 15% and 60% of task switching overheads. We believe that this amount of idle time could be justified by delays in task switching overheads (point a.). On the other hand, for the benchmarks in Group L run with more than four workers a considerable part of non-primary idle time is caused by the inability of the scheduler to explore very fine grained parallelism (point b. above), as indicated by significant decrease in idle time for these benchmarks when the frequency of `CHECK PARALLEL` is set to 1.

Let us examine some differences in the overheads within the groups. The averages for the total of task switching overheads in Table 3.5 do not show a significant variation between the groups. The totals for individual benchmarks, however, vary considerably, ranging from 0.48 msec (`db4`) to 1.34 msec (`farmer`) for 11 workers. Detailed analysis of our performance data [9] shows that the time needed for major scheduling events (such as `MAKE PUBLIC`, `FIND WORK`) is fairly constant[4]. The *frequency* of these events (i.e. how often does a specific event occur during a task) varies considerably, and so this is the main cause of the differences in total task switching costs.

The frequency of scheduling events (other than backtracking) is strongly related to the branching factor of the public tree, i.e. the average number of branches below public nodes. As confirmed by the measurements, the bigger the number of branches, the greater is the chance that public backtracking leads to a live node, thus reducing the need for going through the `FIND - RELEASE - GO TO WORK` loop.

Let us summarise the results of the this section. The total run time of a program in Aurora can be split up into the following components:

$$run\_time = execution\_time + task\_switching\_overheads + idle\_time$$

where

$$execution\_time = sequential\_execution\_time + execution\_overheads$$

The *execution_overheads* are proportional to the *sequential_execution_time* (roughly 25-30%, except for the very fine granularity benchmarks). The *task_switching_overheads* are proportional to the number of tasks: 0.5-1.5 msec per task (for 11 workers), depending upon the branching factor of the search tree. Finally *idle time* is influenced by several factors: the theoretical amount of parallelism available in the program (cf. primary idle time), delays in creation of work due to the task switching overheads and the granularity of available parallelism (too fine grained parallelism is too expensive to be exploited with the current scheduler). The first two factors seem to be relevant to the medium-coarse granularity programs (the non-primary part of the idle time being about 15-60% of task switching overheads), while for the fine granularity examples the third factor gains crucial importance.

Let us now turn to the question of discrepancies between simulator predictions and actual measurements. As shown in Table 3.5 the group average of total task switching overheads goes up to the equivalent of about 6 Prolog calls (reaching 7 calls for some of the individual benchmarks). At the same time the speed-ups are significantly worse than the ones predicted by the simulator for the overhead of 8 resolution time units (Table 3.2). One of the main reasons for this difference in speed-ups is the fact that the simulator has a different notion of a *task*: when a worker backtracks to a live shared node and is able to pick up a new branch at that node, then the work done on the new branch will not be treated as a new task by the simulator. More importantly the basic scheduling algorithm causes some subtrees that are never shared to be split into several tasks. This is because each worker tries to keep a live node public on its branch and so may make nodes public unnecessarily. A refinement of the scheduling algorithm, lazy release of work, which aims at avoiding this behaviour, is outlined in Section 3.7.

3.6 Locking and moving overheads

Locks are used within the Aurora implementation to synchronise various activities of workers exploring the Prolog search tree in parallel. Locking is needed when extending or shrinking the public part of the tree and also when updating various subfields of the scheduler data structures, e.g. bit-maps in the nodes, interrupt message areas in the worker data structures etc. The standard locking macros are used as provided by the DYNIX operating system. These macros involve a busy waiting loop if the lock is held by another worker.

[4] with the exception of `GO TO WORK` for `parse5`, see Section 3.6 for details.

Bench-mark Groups	MAKE PUBL.	BACK-TRACK	FIND	REL.	GO TO	OTHER	IDLE	TOTAL	TOTAL LOCKING TIME as % of OVERHEAD TIME	TOTAL LOCKING TIME as % of FULL RUN TIME
	Average locking time (μsec)			WORK						
Group H										
w = 1	4	4						4	4.95%	0.02%
w = 4	4	7	5	6	6	7	7	6	6.00%	0.19%
w = 8	4	8	5	8	5	9	16	7	6.51%	0.51%
w = 11	4	11	6	8	6	9	16	8	6.94%	0.82%
Group M										
w = 1	5	4						5	4.29%	0.02%
w = 4	4	5	5	6	6		7	5	6.07%	0.68%
w = 8	4	8	6	7	6		15	7	6.86%	1.94%
w = 11	4	8	7	8	7		18	8	7.02%	2.59%
Group L										
w = 1	4	4						4	4.11%	0.04%
w = 4	4	8	6	6	7	10	7	7	7.10%	2.68%
w = 8	4	12	8	7	9	14	14	10	7.31%	4.65%
w = 11	4	14	10	7	9	21	21	12	6.96%	5.13%

Table 3.7: Locking statistics

In the instrumented version of Aurora the time needed for acquiring locks has been accumulated separately within each of the activities of Fig. 3.1. This proved to be extremely useful in identifying those activities where the congestion of workers competing for locks was the biggest. Some algorithms within the Manchester scheduler have been rewritten to avoid holding locks for unnecessarily long time and a restricted use has been made of the specific atomic operations provided by the 80386 processor to avoid locking. This helped to reduce the average locking time to below 10 μsec for most of examples as shown in Table 3.7.

The first part of Table 3.7 gives the average locking time within each of the main activities of Fig. 3.1 and the average of all locking times (the `TOTAL` column). Some fields are left empty – this means there are no locking operations within the given activity for the given number of workers. The minimal time required to grab a lock is about 4 μsec, the difference between that and the figures shown is the time spent in the busy waiting loop. The figures in the table indicate that there is still some congestion in backtracking and in the idle activities.

In the second part of Table 3.7 the percentage of total locking time is shown both within the total (task switching and idle) overhead time and within the full run time. It is quite reassuring to note that even in the worst case of Group L, w=11, just 5% of total run time is spent in locking, and only $\frac{2}{3}$ of that is spent in the busy waiting loop (8 μsec out of the average 12 μsec locking time).

Table 3.8 shows various data on migration (binding array update) costs. For the purpose of this table `parse5` has been excluded from Group M and has been shown separately, as it exposes one of the weak points of the Manchester scheduler.

The first part of the table shows the percentage of time spent updating binding arrays within various overhead categories, of which, naturally, `GO TO WORK` is the most significant. The next two columns refer to this specific category: the number of moves (number of nodes stepped through) and the migration time for an average `GO TO WORK` event. This is followed by the number of bindings that have to be handled during one average move, while the last two columns give the percentage of migration time within total overhead time and within the full run time.

The group averages in Table 3.8 show that the overall effect of migration costs is not very significant: it accounts for at most 10% of total overheads and at most 5% of full run time. The data for `parse5` is more alarming: almost 30% of overhead time is due to migration costs. The cause of this lies in a rather special shape of the search tree of `parse5`: it has two long branches with rather small amounts of work appearing on both of these from time to time. A worker looking for work may find that the only piece of work available at that very moment is on the other branch, in which case it will pick up that piece of work, irrespective of

Bench-mark Groups	MIGRATION TIME within BACK-TRACK	GO TO WORK	OTHER	IDLE	GO TO WORK number of MOVES	GO TO WORK MIGR. TIME (msec)	MIGR. BINDs per MOVE	TOTAL MIGR. TIME as % of OVERHEAD	TOTAL MIGR. TIME as % of FULL RUN TIME
	% of full time of the overhead								
Group H									
w = 1	5.12%							2.62%	0.00%
w = 4	7.50%	30.58%	0.50%	1.99%	4.40	0.06	0.46	4.49%	0.10%
w = 8	7.81%	31.22%	0.42%	1.42%	6.41	0.07	0.49	4.94%	0.28%
w = 11	7.77%	31.04%	0.62%	1.15%	6.77	0.08	0.49	5.14%	0.43%
Group M'									
w = 1	3.17%							1.75%	0.01%
w = 4	8.74%	41.48%		5.63%	3.94	0.10	1.34	8.03%	0.98%
w = 8	8.57%	42.65%		4.71%	6.47	0.17	1.57	10.41%	3.29%
w = 11	8.14%	41.99%		4.32%	7.60	0.20	1.56	10.37%	4.18%
parse5									
w = 1	2.79%							1.56%	0.00%
w = 4	13.50%	61.21%		13.63%	8.54	0.29	3.66	19.00%	1.48%
w = 8	12.41%	60.76%		10.91%	12.53	0.44	3.66	22.63%	5.94%
w = 11	9.59%	62.35%		15.99%	14.20	0.68	3.55	28.01%	9.36%
Group L									
w = 1	2.60%							1.42%	0.01%
w = 4	9.05%	43.21%	0.97%	5.79%	4.00	0.10	1.71	8.78%	3.08%
w = 8	8.81%	42.28%	1.19%	3.08%	5.28	0.14	1.73	7.30%	4.50%
w = 11	8.28%	42.67%	1.17%	2.17%	5.22	0.15	1.78	5.67%	4.09%

Table 3.8: Migration costs

the distance. A more refined scheduling algorithm could start moving towards a very distant piece of work without actually reserving it and could reverse its decision if a new piece of work appears nearby.

3.7 Tuning the Manchester scheduler

Three refinements of the Manchester scheduling algorithm, aimed at increasing task size and reducing task switching costs, have been evaluated in the expanded version [9] of the present paper. Two of these have been found beneficial: lazy release of work (nodes not being made public if there are no idle workers) and straightening (the tree structure being simplified by removing dead non-fork nodes as described in [11]). There is up to 10% improvement in speed after these two refinements have been applied as shown in Table 3.9, in spite of some shortcomings of their present implementation.

3.8 Conclusions

The Manchester version of the Aurora or-parallel Prolog system has been evaluated on a diverse set of benchmarks for up to eleven processors. The main components of the implementation have been identified and the system has been instrumented to collect both time and frequency data for these components.

The analysis of the performance data confirms the correctness of main design decisions of Aurora. The SRI binding scheme is shown to impose a constant overhead of about 30% on the sequential execution time. The migration costs of updating binding arrays, which were feared to be a major source of overhead, proved to constitute a rather small proportion of total overheads for most of the benchmarks.

The costs associated with synchronisation using locks have also been examined. The profiling data has been used to locate and eliminate those parts of the scheduling algorithm where the congestion of workers

Goals * repetitions	Workers 1	4	8	11
8-queens1	11.47	2.87*(4.00)*	1.45*(7.91)*	1.07*(10.7)*
8-queens2	32.72	8.30*(3.94)*	4.17*(7.85)*	3.06*(10.7)*
tina	23.74	6.22*(3.82)*	3.26*(7.28)*	2.49*(9.54)*
sm2 *10	13.56	3.45*(3.93)*	1.80*(7.53)*	1.35*(10.0)*
AVERAGE		*(3.92)*	*(7.64)*	*(10.2)*
parse2 *20	10.18	3.21*(3.17)*	2.18*(4.68)*	1.97*(5.17)*
parse4 *5	9.47	2.58*(3.67)*	1.83*(5.18)*	1.49*(6.36)*
parse5	6.72	1.90*(3.54)*	1.28*(5.26)*	1.08*(6.22)*
db4 *10	3.62	0.98*(3.70)*	0.55*(6.60)*	0.44*(8.25)*
db5 *10	4.42	1.18*(3.75)*	0.68*(6.50)*	0.54*(8.17)*
house *20	9.13	2.54*(3.60)*	1.46*(6.26)*	1.18*(7.76)*
AVERAGE		*(3.57)*	*(5.74)*	*(6.99)*
parse1 *20	2.72	0.99*(2.75)*	0.83*(3.28)*	0.87*(3.13)*
parse3 *20	2.32	0.93*(2.50)*	0.80*(2.91)*	0.80*(2.91)*
farmer *100	5.33	2.58*(2.07)*	2.49*(2.14)*	2.47*(2.16)*
AVERAGE		*(2.44)*	*(2.78)*	*(2.73)*

Table 3.9: Timing results with straightening and lazy release

competing for locks was highest. In the new system, locking accounts for only about 6-7% of total overhead time, increasing only slightly for more workers.

There is a significant administrative overhead associated with task switching, which is equivalent to 4–7 Prolog calls per task. This puts a relatively high burden on programs with small task size. Furthermore there is a significant amount of idle time which cannot be explained by lack of parallelism. We believe that this part of the idle time is caused by delays in creation of work due to earlier task switching overheads. Consequently task switching, and especially public backtracking, seems to be the major source of overhead in the or-parallel execution.

Several ways of reducing the task switching costs can be envisaged, in addition to the refinements outlined in Section 3.7. The unit cost of task switching could be reduced by simplifying and tuning the scheduling algorithm. The engine-scheduler interface could be modified to allow the engine to explore several branches of a public node within a single task, avoiding some administrative costs of exiting and reentering the engine. Finally, the recent results of the BC-machine project [1] indicate that dispatching on bottommost (i.e. releasing work at the youngest node on a worker's stack) may result in significant reduction in task switching costs.

3.9 Acknowledgements

The author would like to thank his colleagues in the Gigalips project at Argonne National Laboratory, the Swedish Institute of Computer Science, and the University of Bristol. Thanks are due in particular to Alan Calderwood and David Warren.

The work was supported by the UK Science and Engineering Research Council and by the ESPRIT project PEPMA.

References

[1] Khayri Ali. Or-parallel execution of Prolog on BC-Machine. SICS Research Report, Swedish Institute of Computer Science, 1987.

[2] Per Brand. Wavefront scheduling. Internal Report, Gigalips Project, 1988.

[3] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Logic Programming: Proceedings of the Fifth International Conference*, pages 1590–1605. The 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. The MIT Press, June 1989.

[5] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*, pages 819–830. ICOT, Tokyo, Japan, November 1988.

[6] Kish Shen. An investigation of the Argonne model of or-parallel Prolog. Master's thesis, University of Manchester, 1986.

[7] Kish Shen. Personal communication, October 1988.

[8] Péter Szeredi. More benchmarks of Aurora. Internal Report, Gigalips Project, March 1988.

[9] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. Technical Report TR-89-14, University of Bristol, 1989.

[10] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. The MIT Press, October 1989.

[11] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

Chapter 4

Flexible Scheduling of Or-parallelism in Aurora: The Bristol Scheduler[1]

Anthony Beaumont, S Muthu Raman[2], Péter Szeredi[3] and David H D Warren

Department of Computer Science, University of Bristol,
Bristol BS8 1TR, U.K.

Abstract

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, based on the SRI model of execution. It consists of a Prolog engine based on SICStus Prolog and several alternative schedulers. The task of the schedulers is to share the work available in the Prolog search tree

This paper describes the Bristol scheduler. Its distinguishing feature is that work is shared at the bottom of partially explored branches ("dispatching on bottom-most"). This can be contrasted with the earlier schedulers, which use a "dispatching on topmost" strategy. We argue that dispatching on bottom-most can lead to good performance, by reducing the overheads of scheduling.

Our approach has been to find the simplest scheduler design which could achieve performance competitive with earlier more complex schedulers. This design gives us a flexibility in deciding strategies for sharing work and allows us to examine ways of improving the performance on both non-speculative and speculative work. We note that in speculative regions the priority of some work is higher than others. We have investigated strategies which help workers to avoid low priority work.

We present the basic design of the Bristol scheduler, discussing the data structures and the main algorithms. We also present performance results for the new scheduler using a number of benchmark programs and large applications. We show that the performance of the Bristol scheduler compares favourably with other schedulers. Our work also shows that special treatment of speculative work leads to improved performance.

Keywords: Implementation, Or-parallelism, Multiprocessors, Scheduling.

4.1 Introduction

Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors, currently running on Sequent and Encore machines. It has been developed in the framework of

[1] This paper has appeared in the proceedings of PARLE'91 [3]
[2] *Visiting from* National Centre for Software Technology, Gulmohar Cross Road 9, Juhu, Bombay 400 049, India
[3] *On leave from* SZKI, IQSOFT, Donáti u. 35–45, Budapest, Hungary

the Gigalips project, a collaborative effort between groups at Argonne National Laboratory, University of Bristol, and the Swedish Institute of Computer Science (SICS). A full description of Aurora can be found elsewhere [9].

Aurora is based on the SRI model [14] in which or-parallel execution of Prolog programs consists of the exploration of a search tree in parallel by a number of *workers*. A worker is defined as an abstract processing agent. During execution, a tree of *nodes* is created, where each node represents a Prolog choicepoint. A worker will begin working on a *task* by taking an alternative from a node, creating a new arc of the tree. The task will be explored in the normal sequential Prolog manner, and will end when the worker runs out of work. The way workers move around the tree and communicate with each other in order to find tasks is determined by some *scheduling strategy*.

Branches of the tree are extended during resolution and destroyed on backtracking. A major problem introduced by or-parallelism is that some variables may be simultaneously bound by workers exploring different branches of the tree. The SRI model dictates that each worker will maintain a *binding array* to hold the bindings associated with its current branch. We can say that a worker is *positioned* at a node, when its binding array holds the bindings associated with the path between the root and the given node. Moving up a branch involves removing bindings, while moving down involves adding bindings to the binding array.

In Aurora the search tree is divided into *public* and *private* regions, the boundary between the two being marked by a *sentry node*. Private regions contain nodes which are explored by a single worker, and for workers to be able to share work at a node, that node has to be made public. Another distinction is that nodes can be either *parallel* or *sequential* through user declarations. Alternatives from sequential nodes can only be executed one at a time. We can also think of each node as being either *live* (i.e. having unexplored alternatives) or *dead* (no alternatives to explore). A node is called a *fork node* if it has more than one child.

An Aurora worker consists of two components: the *engine*, which is responsible for executing the Prolog code and the *scheduler*, responsible for finding work in the tree and for synchronising with other workers. There is a strict interface between these components [13] which enables independent development of different schedulers. Aurora execution is governed by the engine: whenever the engine runs out of work in its private region it will ask the scheduler to find more; a process called *task switching*. The Aurora engine is based on SICStus Prolog version 0.6 which has been extended to comply with both the SRI model and the engine/scheduler interface.

4.2 Scheduling Strategies

We now discuss the problem of finding a new task for a worker which has run out of work. We have already stated that work can only be taken from public nodes, therefore idle workers must find a live, parallel, public node. If we assume that initially all work is public then we could allow idle workers to search the tree to find work. This will allow work to be found from any branch but not without some cost. An idle worker may have to search a large number of nodes before work is found, and also the search will require some synchronisation to avoid searching branches as they are being reclaimed by backtracking workers.

To focus the search into areas of the tree where work may be more likely to be found we could search only those branches which are currently being extended by busy workers. Selected workers could be scanned to assess whether the branch they are working on contains work or not.

We should note however that a branch can be explored quicker by a single worker if that worker keeps the task private, rather than making some or all of it public. This indicates that if all workers are busy then there is no reason to make work public and therefore it would be better to assume that initially all work is private and that workers make work public on demand only.

Following this approach, an idle worker searching for a new task might select a worker which has private work and ask it to share some or all of it. We must remember however that searching for work only on branches which are currently being explored assumes that all live nodes have at least one busy worker positioned below them, if this assumption is not true, some nodes will become inaccessable.

Another consideration is which of the available tasks an idle worker should prefer. When earlier schedulers were designed it was thought that a worker should keep most of its work private to make its task as large as possible. Only the topmost task was made available to other workers. If the busy worker kept the topmost task public and that task was not quickly exhausted then the worker would not be interfered with as it explored the rest of the branch. This is known as *topmost dispatching*.

An alternative strategy investigated in this paper is *bottom-most dispatching* when work is shared at the bottom of partially explored branches and we will discuss this later.

4.2.1 Topmost dispatching schedulers for Aurora

There were three earlier schedulers for Aurora, all using topmost dispatching.

The Argonne scheduler [5] uses local information that is maintained in each node to indicate whether there is work available below the node. Workers search the tree, using this local information to migrate towards regions of the tree where work is available. The workers always take the topmost task from a branch since this is always the first task found as they move down. A bitmap in each node indicates which workers are positioned at or below the node and workers are required to update these bitmaps as they move around the tree. Information in the bitmaps can be used to locate other workers, for example in the case of pruning a subtree it is necessary to inform the workers which are positioned in that subtree that they have been pruned.

The Manchester scheduler [6] tries to match idle workers with the nearest available outstanding task, where "nearness" is measured by the number of bindings to be updated between the worker's current position and the available work. Minimising the distance between worker and task means that the worker will not consider any task below the topmost one on each branch. Again bitmaps are employed to mark the presence of workers on a branch. The Manchester scheduler uses them both for matching idle workers to available work and for locating workers during pruning.

The Wavefront scheduler [4] employs a data structure known as the wavefront which links all the topmost live nodes together. Workers find work by traversing the wavefront. As nodes are exhausted the wavefront is extended to allow access to the next live parallel node.

Topmost dispatching, used by all of these schedulers, has the disadvantage that unless the topmost task is large it will be quickly exhausted and the worker will have to repeat its search for work. This leads to relatively high task switching costs for fine granularity programs and also slows down the busy workers since they have to spend more time maintaining a live public node at the top of their branch.

4.2.2 The Muse Scheduler

Another approach to the or-parallel implementation of Prolog is the Muse system [1][2] which is based on having several sequential Prolog engines, each with local address space and some shared memory space. Workers in Muse copy each other's state rather than sharing it as is the case in Aurora. Potentially increasing the overheads involved in task switching. Therefore Muse requires a way of reducing the frequency of task switches involving copying.

The Muse scheduler uses bottom-most dispatching, so that a busy worker, when interrupted for work, will share all nodes on its branch. This allows an idle worker to begin work at the bottom-most of these nodes. The advantage of this strategy is that once the work at the bottom-most node is exhausted, more work can be found by simply backtracking to the next live parallel node, further up the branch. Backtracking to a public node is more expensive than backtracking to a private one, however these *minor* task switches are much less expensive than the *major* task switches which require a wider search for work. It has been found that bottom-most dispatching can reduce scheduling overheads by increasing the number of minor task switches and reducing the number of major task switches.

To help an idle worker decide which busy worker to interrupt for work, Muse introduces the concept of *richness*. Each branch of the tree has an associated richness, which is an estimate of the amount of work on that branch. In the muse system, richness is based on the number of unexplored alternatives on the branch. An idle worker will choose a busy worker from the subtree below the idle worker's current node, the choice of worker depends on the richness of each busy worker's branch. The idle worker will interrupt the worker which is working on the richest branch, ie. has the most work to share, which will further help in increasing the ratio of minor to major task switches.

4.3 Principles of the Bristol scheduler

In designing the Bristol scheduler we took into consideration the results from performance analyses of earlier schedulers and used this information to try and incorporate the best features of other schedulers into our design.

A performance analysis of the Manchester scheduler [12] indicated that the migration of workers to new tasks was not a significant overhead and that much more important was the administrative overhead associated with task switching, estimated to be equivalent to 4-7 Prolog calls per task. The conclusions of this analysis are that simplifying the scheduling algorithm and tuning the scheduler could reduce the costs of task switching and, more importantly, minimise the number of major task switches. Keeping this in mind we have tried to keep the design of the Bristol scheduler as simple as possible.

One of the requirements of the Bristol scheduler is that it should be flexible enough for us to try different scheduling strategies and this will allow us to compare bottom-most and topmost dispatching using the same scheduler. However, based on the good results of the Muse scheduler, we decided to use bottom-most dispatching as the default strategy. The key overhead in earlier Aurora schedulers is the major task switch. If bottom-most dispatching reduces the number of major task switches, and if minor task switches are not very expensive then the total scheduling overheads will be reduced.

A second reason for using the bottom-most dispatching strategy is its suitability for scheduling *speculative* work. This is illustrated by the following program:

```
p:- condition, !, pred1.
p:- pred2.
```

All work in the second clause is said to be speculative because if *condition* succeeds then the second branch will be pruned away. Intuitively, it would seem better to direct workers to help in evaluating *condition*, rather than *pred2*. Similarly, one would want to give higher priority to work which is further to the left within *condition* [7]. Therefore, in a speculative subtree the deepest work on the left-hand branch is the least speculative and should have the highest priority.

Earlier schedulers could not handle speculative work at all effectively. Our aim is to implement an effective speculative scheduling technique within the Bristol scheduler. The bottom-most dispatching strategy helps in directing workers to deeper regions of the search tree but this is not sufficient on its own as a scheduling strategy since the deeper branches may not be the least speculative. What we require is some way of concentrating workers in the leftmost region of a speculative subtree.

This suggests that rather than rely on taking work from busy workers, idle workers would need to scan a speculative subtree to find the least speculative available work. Our design allows us to experiment with such a strategy.

A problem with bottom-most dispatching is that it increases the size of the public region of the tree and backtracking through this region (*public backtracking*) is more expensive than private backtracking. We will try to analyse the effect of this problem by comparing bottom-most and topmost dispatching strategies using the Bristol scheduler.

4.4 Implementation of the Bristol scheduler

During this section we will discuss some of the issues involved in the implementation of the Bristol scheduler.

4.4.1 Data structures

We include the notion of richness introduced by the Muse scheduler and use an estimate of the number of live nodes on a branch as the richness of each branch. Actually, each node is given a richness which is an estimate of how many live nodes there are above it.

Primarily a worker wants to know if another worker has work available and must be able to send a message to it, for example, to ask for work. In our implementation, each worker has a message area, enabling other

workers to send messages to it, and a record of the richness of its current branch which can be read by other workers.

To give some indication of a worker's position in the tree, each worker has an associated *root* node, which is defined to be the root of the subtree in which that worker is leftmost. Initially, this is equal to the workers' sentry node when it first starts work. The identifier of the worker is stored in its root node, and that worker will be known as the *owner* of the node. All nodes subsequently created will contain a pointer (so called *root pointer*) to that workers' root node.

A worker's root may change due to the actions of other workers in backtracking. When a worker backtracks to a fork node from the first child, leaving another worker below, then the backtracking worker must move the root of the remaining worker up the tree to reflect the change. This is illustrated in Figure 4.1 where worker A backtracks out of the left subtree, leaving worker B as the new leftmost worker (the letters in the root nodes show the identity of the worker they belong to). In this case, worker A will put B's identity into its old root and make that node the new root of worker B. B's old root has its root pointer set to point to the new root.

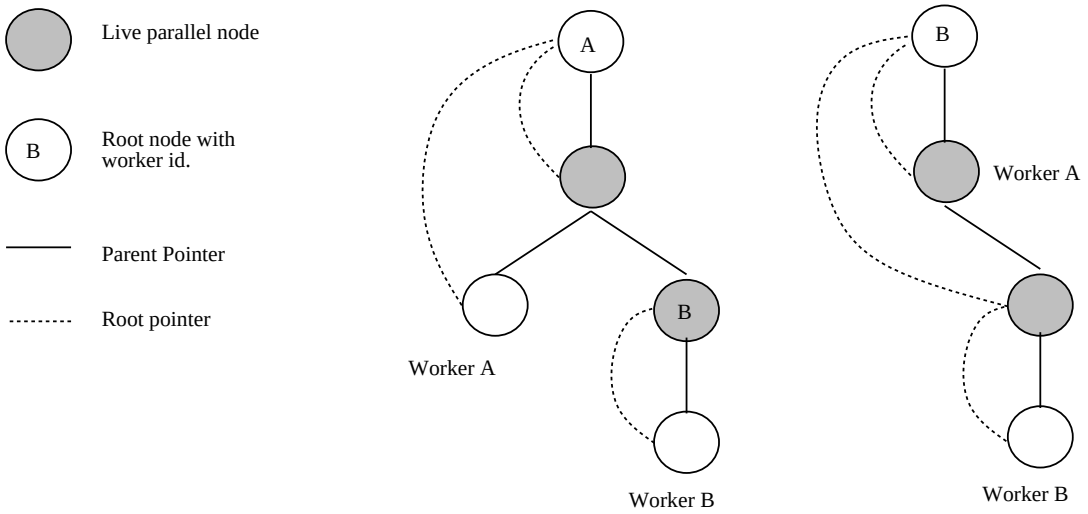

Figure 4.1: Worker A backtracking to a fork node

In a speculative region we will want to find the leftmost (least speculative) task, and therefore will need some way of searching a subtree from left to right. By following the root pointers to a root node and finding its owner, we have a way of finding the leftmost worker in any subtree, and we can also tell if that worker has work available. To continue searching for work, a worker requires some way of finding the bottom of the leftmost branch and moving right. Each worker maintains a pointer to its sentry node which marks the leaf of the public branch. After identifying the owner of a root node and if that worker has no private work then the worker's sentry node pointer is used as a way of gaining access to the bottom of that worker's branch. The owners identity in a root node acts as a *leaf pointer*.

In order to simplify the further search for work the notion of *right pointer* is introduced. This is illustrated in Figure 4.2 where we can see how the tree is organised. The right pointer points to the next sibling, if there is one. For the rightmost child of a live node the right pointer will point to itself, indicating the potential work present there. For the rightmost child of a dead node the right pointer will point upwards to the first ancestor which has either a right sibling or a live parent.

We use a flag to indicate that the right pointer of a node points to a right sibling and not to some other node. Using the root, leaf and right pointers a worker can search around the tree from left to right. This method of linking the nodes of a tree to allow left to right traversal was taken from the data structure used in Andorra-1 for maintaining the goal list [10].

4.4.2 Looking for work

The engine will hand over control to the scheduler when the worker backtracks to its sentry node and the scheduler will be responsible for finding work in the public region of the tree. We are exploring two different

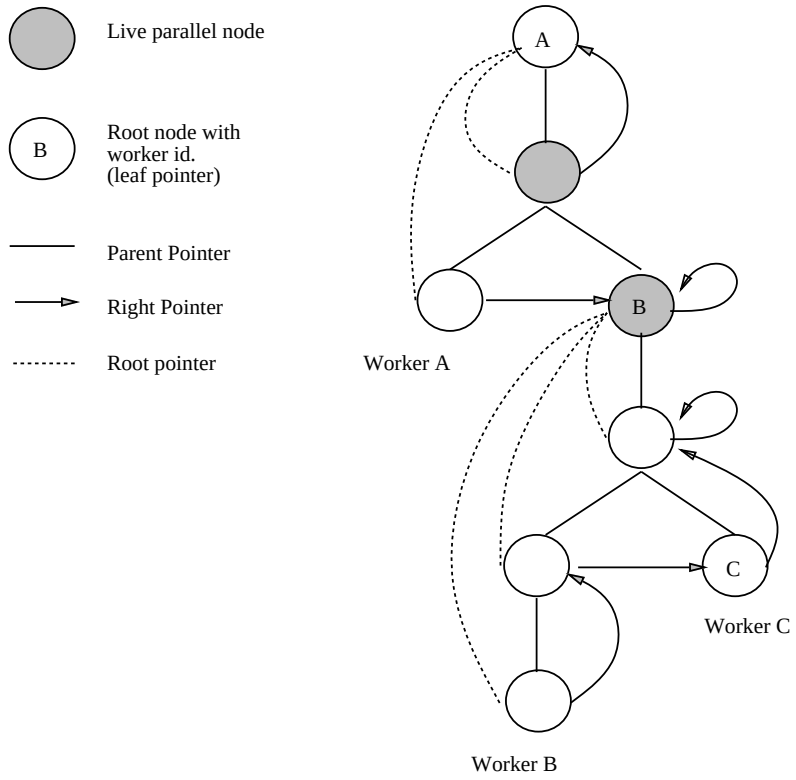

Figure 4.2: Organisation of the tree

strategies which the scheduler will use to find work, the *richest worker* strategy and the *left-to-right search* strategy.

4.4.2.1 The richest worker strategy

Following the richest worker strategy, the scheduler will attempt to find work in two ways; Firstly it will backtrack through the nodes above the worker's current position to see if any work can be found nearby. If a live parallel node can be found then the scheduler takes an alternative from it and returns control back to the engine to restart work. If no work can be found then the other workers will be scanned to see if they have work available. The idle worker will identify the worker which is working on the richest branch and interrupt it for work. That worker will then make all of its private nodes public and the idle worker can begin work at the bottom of that branch. Note that the number of nodes being made public is flexible in that we could limit this number and therefore bottom most dispatching need not necessarily be used.

This strategy has the advantage that work is made public only on demand, so if a worker is not interrupted for work it will not make any nodes public. Workers may have to migrate further when taking work from another worker. We hope that the bottom-most dispatching strategy will minimise the number of major task switches, and most work will be found quickly from a node just above the workers current position.

4.4.2.2 Left to right search

We want to get some indication of how much can be gained by treating speculative work differently from non-speculative work and have explored an alternative strategy, which we call left-to-right search. The assumption behind this strategy is that the whole tree is speculative and that work on the leftmost branch has the highest priority, and the priority of work decreases the further away it is from the leftmost branch. This assumption clearly represents too narrow a view of speculative work; in general all work is non-speculative unless it is in the scope of a pruning operator. `Setof` (and `bagof`) create subtrees which are locally non-speculative even if the setof itself is speculative, ie all branches should have equal priority. We describe a more refined strategy for handling speculative work in Section 4.6.

Related work by Sindaha [11] uses a similar method to search for work in a speculative subtree. However the left to right search is implemented by explicitly linking the sentry nodes of all branches to create a data structure similar to the wavefront. Workers find work by traversing this data structure. This work is not as far advanced as the Bristol scheduler but we hope later to compare this approach with our own.

The left to right search strategy also uses bottom-most dispatching. Finding work will again begin by backtracking through the nodes immediately above the worker's current position to find nearby work. If no work can be found the worker will search the tree from left to right, by following the root, leaf and right pointers. The leftmost worker can be identified as the owner of the root of the tree, the scheduler will search right from the leftmost worker until it finds either a worker with private work which it can share, or a live parallel node.

4.4.3 Side-effects and suspension

A goal of the Aurora system is to implement all standard Prolog built-in predicates, preserving their sequential semantics. To achieve this, we delay the execution of a call to a side-effect predicate until it becomes leftmost in the whole tree. We implement this delay by allowing workers to suspend a branch when executing a side-effect predicate. There are some special predicates, for example those used in the implementation of `setof`, where it is sufficient to ensure that the branch is leftmost within some subtree. Checking whether a worker is leftmost within some subtree is done simply by testing if the worker's root node is at or above the root of the subtree. Asynchronous versions of these side-effect predicates are also provided and these suspend only if they occur within the scope of a cut and may be pruned.

To suspend a branch, all a worker must do is to mark its root node as suspended and make the root of the suspended node point down to the sentry node on the suspended branch. The scheduler will then find a new task for the suspending worker to begin.

We next look at how suspended branches can be restarted. If a worker backtracks out of, and reclaims, the leftmost child of a node, it will check to see if there is a suspended right sibling of that node. If there is, it will delete the suspended flag and proceed to check if the branch has now become leftmost in the subtree in which it was suspended. If this is the case the worker will restart the suspended branch. If the branch cannot be restarted the worker will set the suspended flag in its own root and carry on looking for work. The suspended branch will wait until some other worker notices the new suspended node while backtracking.

4.4.4 Cut and commit

Aurora supports two pruning operators: the conventional Prolog *cut*, which prunes all branches to its right and a symmetric version of cut called *commit*, which prunes branches both to its left and right. To preserve the sequential semantics, a pruning operation will not go ahead if there is a chance of it being pruned itself by a cut with a smaller scope. It may be possible to improve on this and we are investigating the method which has been implemented in Muse where the worker will not suspend the branch but partially do the pruning and leave the rest to be done as and when it ceases to be endangered by the cut.

A pruning operation should suspend only if a branch to its left leads to a cut of smaller scope. But, to determine whether a particular pruning operation should suspend or not we need some information about the presence and scope of cuts in the tree.

The engine-scheduler interface [13] makes the necessary information available to the scheduler and we use it to implement the Bristol scheduler's pruning operators in the following way.

The scheduler decorates the tree with information about the presence and scope of cuts. When a node is created which has parallel alternatives containing cut, then that node is marked as a cut boundary node. Each node contains a cut counter, which indicates the number of cuts in the worker's continuation when the node was created. The worker also keeps a cut counter and this is incremented when a clause containing cuts is entered and decremented whenever a cut is executed. When executing a cut, a worker will check if any of the nodes below the cut boundary have children to the left and are either marked as a cut boundary or have a cut counter greater than the workers current cut counter. If such a left sibling is found then the cut will be suspended. For a more detailed description of decorating the tree with cut scope information the reader is referred elsewhere [7].

We will now look at how pruning is implemented in the Bristol scheduler. A pruning worker will visit each

node below the boundary node of the cut (or commit), first removing all unexplored alternatives at the node's parent, and then pruning all the right siblings of the node. If the pruning operation is commit then left siblings must be pruned too. All siblings except the leftmost will be root nodes.

To prune a sibling, the worker will mark the node as pruned and try to identify the worker which will take responsibility for clearing the pruned subtree away. Unless the sibling node is suspended, this means the worker will interrupt the owner of the node. That worker will then pass on the interrupt to any other workers in the subtree. To prune the leftmost sibling in the case of commit, then the worker must follow the node's root pointer to find its root. If its root is not marked suspended then the owner of the root will be interrupted with the information that its subtree is pruned.

If any of the siblings are suspended, it is not possible to identify workers which are in the subtree below that node. These subtrees must be searched to inform any workers which are working there that they have been pruned, although all the pruning worker will do is to marked them as unsearched. The search will be carried out by one of the pruned workers as it moves out of its own pruned subtree. The pruning worker will only search pruned subtrees if no other worker can be found to do the job for it. In this case it will search until it finds the first worker to be pruned and then that worker will take over any further searching.

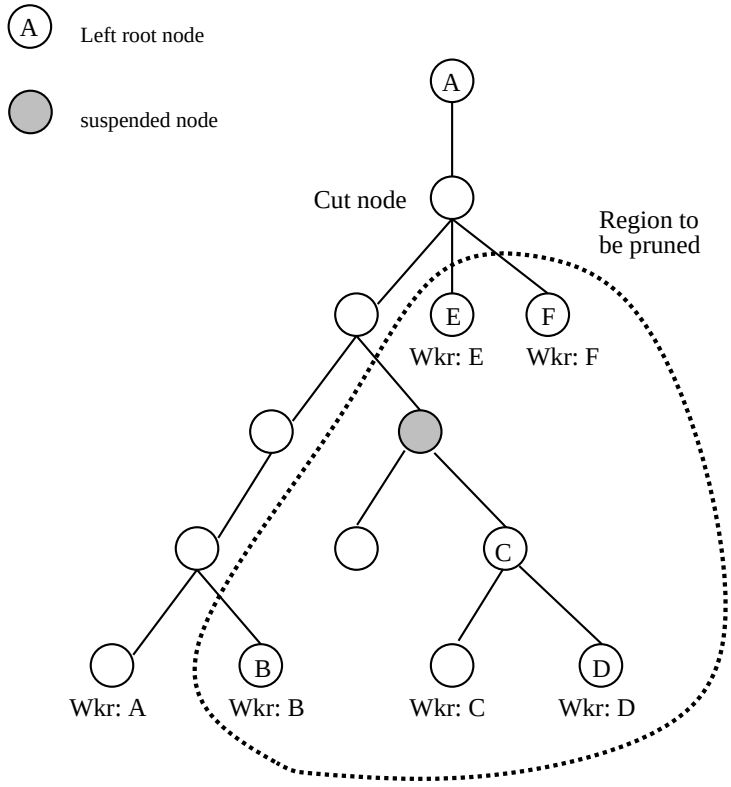

Figure 4.3: Worker A About to perform a cut

Figure 4.3 shows an example where worker `A` wants to perform a cut up to the cut boundary and there are five other workers in the region which is to be pruned. Worker `A` will be able to identify and interrupt workers `B`, `E` and `F`. Worker `B` will search the subtree whose root was suspended at the time worker A did the pruning and will interrupt worker `C`. That worker will in turn interrupt worker `D`. The interrupted workers will then backtrack out of the pruned region and look for work elsewhere.

4.5 Performance results

To assess the performance of various aspects of the Bristol scheduler, we have used a number of benchmarks and application programs, descriptions of which can be found elsewhere [12][8]. All the programs were run on a Sequent Symmetry using the Foxtrot version of Aurora.

To discover which dispatching strategy should be preferred, we have run the Bristol scheduler on a number of programs using 10 workers under both topmost and bottom-most dispatching. We obtained figures on the

frequency and duration of task switches in each program. Task switching begins when a worker backtracks to its sentry node and ends when a new task is found and control passes back to the engine. A distinction is drawn between minor task switches, which end when work is discovered on the same branch, and major task switches, when the worker finds that no work is left on its current branch and it must find work from another part of the tree. Bottom-most dispatching should reduce the number of expensive major task switches, while increasing the number of minor task switches. Table 4.1 shows for each program the average number of task switches made by each worker, and also gives the average duration of each task switch in microseconds.

	Number				Average duration			
	Bottom-most dispatching		Topmost dispatching		Bottom-most dispatching		Topmost dispatching	
Program	Major	Minor	Major	Minor	Major	Minor	Major	Minor
parse1	7.7	12.5	8.5	11.5	2315	310	2037	232
parse2	11.0	26.0	14.2	19.7	1554	219	1377	187
parse3	7.2	12.0	7.7	10.5	2431	209	2342	202
parse4	12.2	47.2	34.7	46.5	1929	229	1560	212
parse5	13.0	87.2	58.7	75.0	2114	213	1707	187
db4	4.5	21.5	5	14.7	1808	196	1386	175
db5	5.0	25.7	6.5	17.7	1465	191	1351	170
farmer	3.7	4.7	4.0	4.2	3298	192	3596	180
house	4.2	13.2	4.0	3.2	1756	255	1873	184
8queens1	1.5	9.25	10.2	10.2	2623	247	1293	204
8queens2	1.7	18.7	16.0	15.5	2674	217	1096	165
tina	21.5	92.7	17.7	52.7	3925	208	4072	168
saltmustard	1.0	3.7	1.0	1.5	2256	231	2276	190
protein	12.0	163.0	74.0	232.0	2804	103	1264	97
warplan	12.7	60.0	30.5	37.2	5347	247	2864	209

Table 4.1: TABLE SHOWING THE AVERAGE NUMBER OF TASK SWITCHES MADE BY EACH WORKER AND THE AVERAGE DURATION IN MICROSECONDS, OF EACH TASK SWITCH

We find that the number of major task switches is significantly reduced while the number of minor task switches is slightly increased. Also the average duration of both major and minor task switches tends to increase somewhat. This reflects the increased size of the public region of the tree, under bottom-most dispatching there will be more public backtracking to be done before the worker eventually finds a new task.

To get a clearer picture of how much time was spent in task switching we computed the percentage of total time, which on average, each worker spent in task switching. This information is shown in Figure 4.4. We can see from this graph that bottom-most dispatching leads to less time spent task switching in all but five of our programs. From this we can conclude that in general it will be better to employ bottom-most dispatching, although there will be some occasions when we will lose out. It may eventually be possible for the scheduler to recognise which dispatching strategy should be used but currently we do not have enough information to implement this.

We next compare the Bristol scheduler with the Manchester scheduler, a topmost dispatching scheduler which is the most developed of the existing Aurora schedulers. The results, given in Table 4.2, reflect fairly closely what we would expect from the comparison of topmost and bottom-most dispatching in the Bristol scheduler, although the Manchester scheduler performs better on a couple of the benchmarks where we might have predicted equal or better performance by the Bristol scheduler. This may be partly due to the Manchester scheduler's strategy of matching idle workers to the nearest available work whereas, with the Bristol scheduler, this does not happen.

To obtain a comparison of two similar bottom-most dispatching schedulers, we have compared Aurora under the Bristol scheduler with Muse 0.6 (Figure 4.3). This version of Muse supports some form of delayed release which is not yet supported by the Bristol scheduler. Also, the muse system has been compiled using the GNU C compiler, which allows use of inline declarations, while the Bristol scheduler was not.

Generally the Bristol scheduler produces better speedups than the Muse scheduler, but this is generally

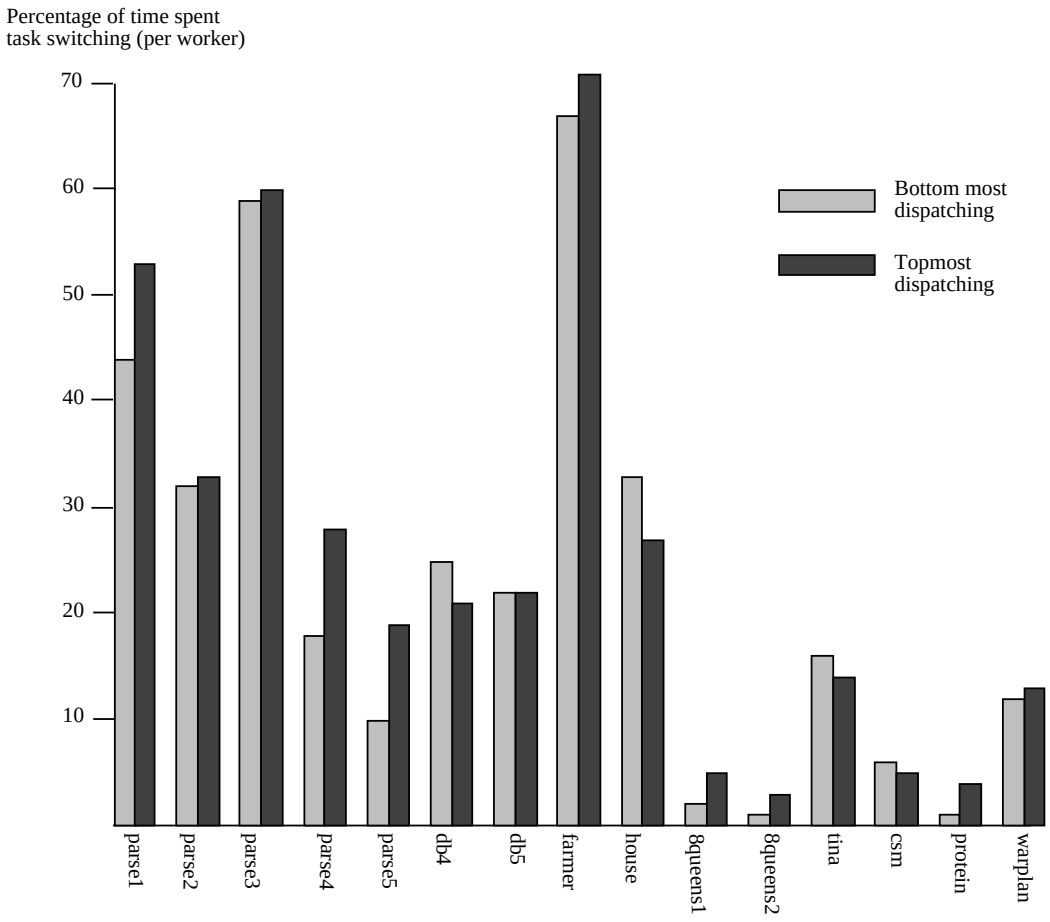

Figure 4.4: Graph of the percentage of total time each worker spends in task switching

somewhat outweighed by the faster engine performance of Muse. Muse's faster engine performance is due to the overhead of the SRI model in Aurora which adds about 25% to the single worker runtime. This is much greater than the corresponding overhead in Muse which is around 5%.

The next part of our performance analysis looks at our strategies for handling speculative work. The left to right search strategy is used here to find out what the benefits might be of treating speculative work differently and what the overheads are. We will later propose a better strategy for adapting the Bristol scheduler for speculative work. We first look at the overheads of the left to right search strategy in finding all solutions. Table 4.4 shows that the 10 worker performance is degraded by about 11%, and this is due both to the extra synchronisation required in searching for work in this manner, and also due to the fact that taking the leftmost available work may not give the worker access to as many live nodes as the deepest available work would have done. Therefore the number of major task switches is increased.

We now take a number of application programs and look at the speedups obtained when finding the first (leftmost) solution, comparing the Manchester scheduler with three versions of the Bristol scheduler; richest worker, an improved version of richest worker where idle workers try to find work from the leftmost worker before trying the richest worker and the left-to-right search. The results are shown in Table 4.5. Since speculative computation always gives some variation in runtimes, depending on how well the workers were utilised during the computation, we present the speedups a ranges of best–worst. The results are given in Table 4.5, and show that when the computation involves speculative work, the bottom-most dispatching strategies all perform better than the Manchester scheduler. The improved richest worker strategy gives some improvement over the original richest worker strategy but the left-to-right search gives the best performance.

	Bristol Scheduler		Manchester Scheduler	
Goals [*Times]	1wkr	10wkrs	1wkr	10wkrs
parse1 *20	1.95	0.74(2.62)	1.94	0.81(2.38)
parse2 *20	7.37	1.50(4.90)	7.35	1.69(4.34)
parse3 *20	1.68	0.67(2.49)	1.66	0.72(2.29)
parse4 *5	6.85	1.04(6.57)	6.80	1.24(5.47)
parse5	4.96	0.64(7.78)	4.79	0.85(5.61)
db4 *10	3.11	0.44(7.06)	3.06	0.41(7.53)
db5 *10	3.79	0.52(7.30)	3.72	0.51(7.34)
farmer*100	3.77	1.95(1.93)	3.80	2.26(1.68)
house*20	5.55	0.94(5.88)	5.17	0.85(6.05)
8-queens1	8.48	0.88(9.65)	8.28	0.83(9.93)
8-queens2	21.89	2.21(9.91)	21.66	2.16(10.0)
tina	19.72	2.14(9.23)	18.77	1.99(9.41)
sm2 *10	11.65	1.32(8.82)	11.27	1.23(9.19)
protein	28.56	3.01(9.49)	27.66	2.94(9.41)
warplan (blocks)	2.73	0.33(8.27)	2.50	0.38(6.58)
warplan (strips)	42.44	4.62(9.19)	40.40	4.46(9.06)
AVERAGE		(6.94)		(6.64)

Table 4.2: Table comparing Aurora under the Bristol and Manchester schedulers (runtimes in seconds with speedups in brackets)

4.6 A strategy for scheduling speculative work

In order to improve on our relatively crude left to right search strategy for speculative work, we have designed a better strategy which will be implemented in a future version of the Bristol scheduler.

For this more general strategy, we no longer treat the whole tree as being speculative but allow for intermingling of speculative and non-speculative subtrees. We will want to distribute workers evenly among speculative subtrees so as not to focus too many resources into one particular subtree. We will also want workers to be able to reassess how speculative their current branch is. For example, a task which was the least speculative of a particular subtree at one moment may later become the most speculative if branches appear to its left. Workers should be able to suspend such a task in favour of a task on one of the higher priority branches to its left, an operation known as *voluntary suspension*. We believe that voluntary suspension is crucial for effective handling of speculative work.

Since it is very difficult to compare the speculativeness of two tasks in separate speculative subtrees we will not allow workers to move from one speculative subtree to another.

Our strategy can then be summarised as follows:

- First try to obtain non-speculative work.
- If only speculative work exists then find work from the speculative subtree containing the smallest number of workers.
- Always search speculative subtrees from left to right.
- Allow workers to periodically consider voluntary suspension of speculative work, in order to find less speculative work.

4.7 Conclusions

We presented a simple, flexible scheduler based on the principle of "dispatching on bottom-most". We have described the algorithms for finding work, public backtracking, pruning and suspension. The current

	Aurora		Muse	
Goals [*Times]	1wkr	10wkrs	1wkr	10wkrs
parse1 *20	1.95	0.74(2.62)	1.58	0.58(2.72)
parse2 *20	7.37	1.50(4.90)	5.89	1.19(5.03)
parse3 *20	1.68	0.67(2.49)	1.36	0.60(2.27)
parse4 *5	6.85	1.04(6.57)	5.53	0.82(6.74)
parse5	4.96	0.64(7.78)	3.91	0.51(7.67)
db4 *10	3.11	0.44(7.06)	2.38	0.35(6.80)
db5 *10	3.79	0.52(7.30)	2.91	0.42(6.93)
farmer*100	3.77	1.95(1.93)	3.12	1.90(1.64)
house*20	5.55	0.94(5.88)	4.35	0.89(4.89)
8-queens1	8.48	0.88(9.65)	6.64	0.70(9.49)
8-queens2	21.89	2.21(9.91)	17.14	1.77(9.68)
tina	19.72	2.14(9.23)	14.79	1.66(8.91)
AVERAGE		(6.28)		(6.06)

Table 4.3: Aurora under the Bristol scheduler, compared with Muse (runtimes in seconds with speedups in brackets)

implementation supports the full Prolog language.

We have presented figures to show that bottom-most dispatching generally produces better performance in Aurora than topmost dispatching, since it decreases the duration of time workers spend in task switching.

The results from running benchmark and application programs show that it is possible to get very good speedups for non-speculative computation from the Bristol scheduler using the richest worker strategy. Comparing that version of the Bristol scheduler with the Manchester scheduler we note that the Bristol scheduler's bottom-most dispatching strategy pays off on the parsing examples where the search trees are deep and narrow. The Manchester scheduler performs better on those examples where the search tree is shallow and broad.

Speedups from the Bristol scheduler are generally better then those obtained from the Muse system, although that system obtains somewhat better overall speed because of the lower overhead involved in adapting Sicstus Prolog to the Muse model.

When working on programs with large amounts of speculative work we can benefit by employing a strategy which prefers to schedule work on left of the speculative region. We can conclude that even though there is an overhead associated with using the left to right search strategy, which is due to the need for synchronisation during the search, we can benefit by using it to schedule work from regions where work on the left side is of higher priority.

We have described a general strategy for handling speculative work, which, based on the results presented here, we believe will give improved performance on speculative work. Our future work will center on implementing this strategy and analysing its performance.

4.8 Acknowledgements

The Authors are indebted to other members of the Gigalips project for careful reading and invaluable comments on this paper, to Mats Carlsson for his work on The Aurora engine and interface, to Bogdan Hausman for his work on speculative scheduling, and to Khayri Ali and Roland Karlsson for their comments and for providing the benchmark timings from Muse.

This work was supported by ESPRIT projects 2471 ("PEPMA") and 2025 ("EDS"). S Muthu Raman was supported by a UN Development Programme Fellowship.

		Richest	Left to
		worker	right search
Goals [*Times]	1wkr	10wkrs	10wkrs
parse1 *20	1	2.62	2.44
parse2 *20	1	4.90	4.14
parse3 *20	1	2.49	2.24
parse4 *5	1	6.57	5.71
parse5	1	7.78	6.79
db4 *10	1	7.06	6.78
db5 *10	1	7.30	6.65
farmer*100	1	1.93	1.68
house*20	1	5.88	4.44
8-queens1	1	9.65	8.83
8-queens2	1	9.91	9.43
tina	1	9.23	7.89
sm2 *10	1	8.82	7.77
AVERAGE		6.47	5.75

Table 4.4: Speedups for different scheduling strategies (Bristol scheduler)

	Aurora	Scheduling strategy			
	one	Manchester	richest	leftmost	left to
Application	worker	scheduler	Worker	then richest	right search
Protein	1	2.90–2.65	2.30–1.96	3.28–2.97	4.46–4.36
Puzzle	1	1.13–1.09	1.33–1.25	2.64–1.77	6.10–5.06
Warplan	1	1.15–1.08	1.11–1.06	1.12–1.10	1.57–1.36
16Queens	1	1.05–1.05	3.35–2.31	3.38–3.30	6.40–3.78
triangle	1	6.44–6.00	7.06–6.60	7.20–6.60	7.68–7.34
tina	1	4.56–4.41	4.56–4.22	4.70–4.48	4.86–4.63
Average	1	2.87–2.71	3.28–2.90	3.72–3.37	5.18–4.42

Table 4.5: Speedups (best–worst) with 10 workers finding the first solution

References

[1] Khayri Ali. *Or-parallel execution of Prolog on BC-Machine*. SICS Research Report, Swedish Institute of Computer Science, 1987.

[2] Khayri A. M. Ali and Roland Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceedings of the North American Conference on Logic Programming*, MIT Press, October 1990.

[3] 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*, Springer Verlag, June 1991.

[4] Per Brand. Wavefront scheduling. 1988. Internal Report, Gigalips Project.

[5] 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.

[6] 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.

[7] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[8] Feliks Kluźniak. *Developing Applications for Aurora.* Technical Report TR-90-17, University of Bristol, Computer Science Department, August 1990.

[9] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[10] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, 1991.

[11] Raed Sindaha. Scheduling speculative work in the Aurora or-parallel Prolog system. March 1990. Internal Report, Gigalips Project, University of Bristol.

[12] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732, MIT Press, October 1989.

[13] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing engines and schedulers in or-parallel prolog systems. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, Springer Verlag, June 1991.

[14] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

Chapter 5

Interfacing Engines and Schedulers in Or-Parallel Prolog Systems[1]

Péter Szeredi[2], Rong Yang

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Mats Carlsson

Swedish Institute of Computer Science
P.O. Box 1263
S-164 28 Kista, Sweden

Abstract

Parallel Prolog systems consist, at least conceptually, of two components: an engine and a scheduler. This paper addresses the problem of defining a clean interface between these components. Such an interface has been designed for Aurora, a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors.

The practical purpose of the interface is to enable different engine and scheduler implementations to be used interchangeably. The development of the interface has, however, contributed in great extent to the clarification of issues in exploiting or-parallelism in Prolog. We believe that these issues are relevant to a wider circle of research in the area of or-parallel implementations of logic programming.

We believe that the concept of an engine-scheduler interface is applicable to a wider range of parallel Prolog implementations. Indeed, the present interface has been used in the Andorra-I system, which supports both and- and or-parallelism.

Keywords: Or-Parallel Execution, Multiprocessors, Implementation Techniques, Scheduling.

5.1 Introduction

Parallel Prolog systems consist, at least conceptually, of two components: an *engine*, which is responsible for the actual execution of the Prolog code, and a *scheduler*, which provides the engine component with work. This paper addresses the problem of defining a clean interface between these components. We focus on a particular interface which has evolved within the implementation of an or-parallel Prolog system, Aurora. The interface has successfully been used to connect the Aurora engine with four different schedulers. It has subsequently been applied in the implementation of the and-or-parallel language Andorra-I, thus proving that its generality extends beyond or-parallel Prolog.

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multipro-

[1] This paper has appeared in the proceedings of PARLE'91 [16]
[2] *On leave from SZKI IQSOFT, Donáti u. 35-45, Budapest, Hungary.*

cessors, based on the SRI model of execution [17], and currently running on Sequent and Encore machines. It has been developed in the framework of the Gigalips project [11], a collaborative effort between groups at the Argonne National Laboratory in Illinois, the University of Bristol (previously at the University of Manchester) and the Swedish Institute of Computer Science (SICS) in Stockholm.

The issue of defining a clear interface between the engine and scheduler components of Aurora was raised in the early stages of the implementation effort. Ross Overbeek made the first attempt to formulate such an interface and Alan Calderwood produced the version [7] used in the first generation of Aurora (based on SICStus Prolog version 0.3).

A fundamental revision of the interface was necessitated by several factors. Performance analysis work on Aurora [14] has shown that some unnecessary overheads are caused by design decisions enforced by the interface. Development of new schedulers and extensions to existing algorithms required the interface to be made more general. The Aurora engine has also been rebuilt on the basis of SICStus Prolog version 0.6.

The new interface, described in the present paper, is part of the second generation of Aurora. The major changes with respect to the previous interface are the following:

- execution is governed by the engine, rather than the scheduler;
- the set of basic concepts has been made simpler and more uniform;
- several potential optimisations are supported;
- the interface is extended to support transfer of information related to pruning operators [10].

The paper is organised as follows. Section 5.2 summarises the SRI model and defines the necessary concepts. Section 5.3 gives a top level view of the interface. Section 5.4 presents the data structures involved in the interface, while Sections 5.5 and 5.6 describe engine-scheduler interactions in various phases of Aurora execution. Section 5.7 shows the extensions: handling of pruning information and various optimisations. Section 5.8 discusses the major issues involved in implementing the Aurora engine side of the interface. Section 5.9 describes how the interface was utilised to introduce or-parallelism into the Andorra-I system [12]. Section 5.10 presents preliminary performance results from the Aurora implementation. We end with a short concluding section.

A complete description of the interface is contained in [15].

5.2 Preliminaries

Aurora is based on the SRI model [17]. According to this model the system consists of several *workers* (processes) exploring the search tree of a Prolog program in parallel. Each node of the tree corresponds to a Prolog *choicepoint* with a branch associated with each alternative clause. A predicate can optionally be declared *sequential* by the user, to prohibit parallel exploration of alternative clauses of a predicate. Corresponding nodes are also annotated as sequential. All other nodes are *parallel*.

As the tree is being explored, each node can be either *live*, i.e. have at least one unexplored alternative, or *dead*. A node is a *fork node* if there are two or more branches below it; otherwise, it is a *nonfork node*. A fork node cannot be sequential. Live parallel nodes, and live sequential nodes with no branches below them, correspond to tasks that can be executed by workers. Each worker has to perform activities of two basic types:

- executing the actual Prolog code;
- finding work in the tree, providing other workers with work and synchronising with other workers.

In accordance with the SRI model each worker has a separate *binding array*, in which it stores its own bindings to potentially shared variables (conditional bindings). This technique allows constant time access to the value of a shared variable, but imposes an overhead of updating the binding arrays whenever a worker has to move within the search tree.

The or-tree is divided into an upper, *public*, part accessible to all workers and a lower, *private*, part accessible to only one worker. A worker exploring its private region does not have to be concerned with synchronisation

or maintaining scheduling data; it can work very much like a standard Prolog engine. The boundary between the public and private regions changes dynamically. It is one of the critical aspects of the scheduling algorithm to decide when to make a node public, allowing other workers to share work at it. In the majority of schedulers, the worker will make his *sentry* node, i.e. his topmost private node, public when all nodes above it have become dead, i.e. have no more alternatives to explore. This means that each worker tries to keep a piece of work on its branch available to other workers.

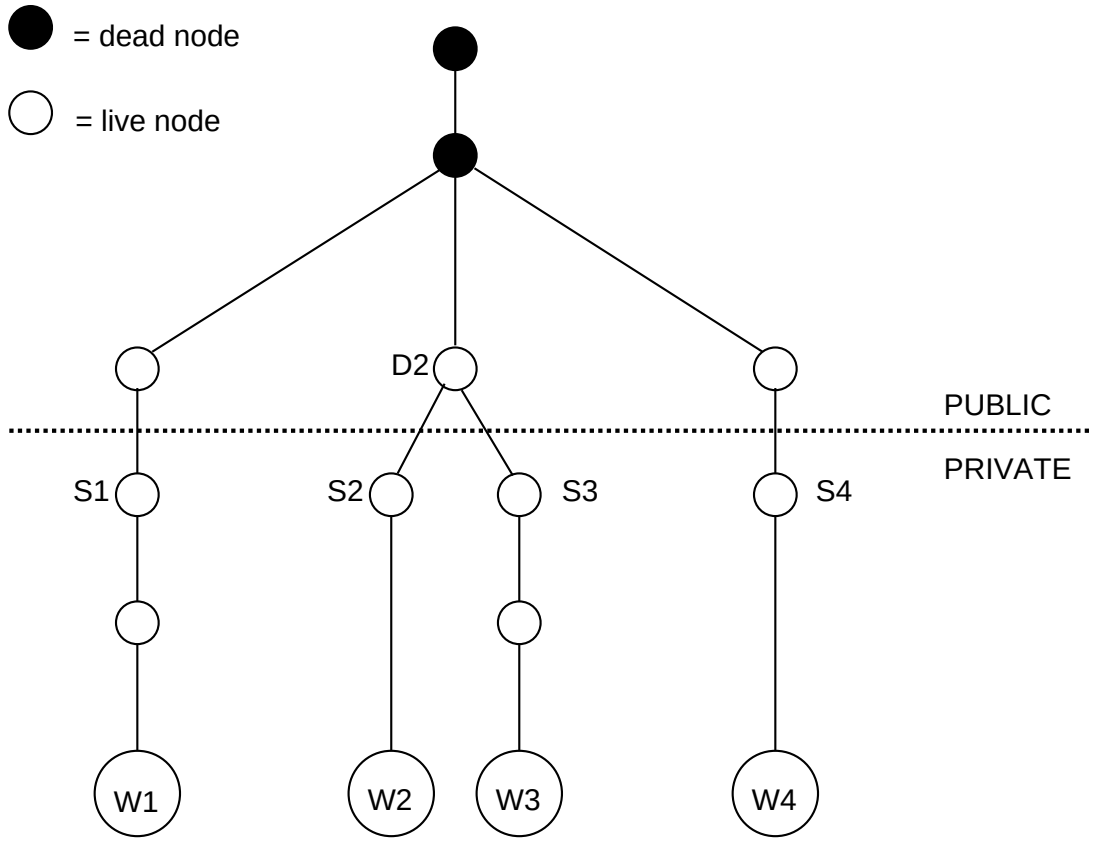

Figure 5.1: The or-tree of the SRI model

For example, in Figure 5.1, an or-tree being explored by four workers (W1–W4) is shown. The workers' sentry nodes are denoted S1–S4. Assume that there is an unexplored alternative at node D2. Now if the branch being explored by worker W1 dies back and W1 takes the alternative at D2, the node D2 will become dead, and the scheduler will normally extend the public region to include nodes S2–S3 so as to keep a piece of work available on every branch.

The exploration by a worker of its private region constitutes that worker's *assignment*, which normally terminates if the worker backtracks into the public part. The assignment terminates prematurely if the branch is *suspended*, or if it is *pruned* by some other worker.

There are three *pruning operators* currently supported by Aurora: the conventional Prolog *cut*, which prunes all branches to its right and a symmetric version of cut called *commit*, which prunes branches both to its left and right. A cut or a commit must not, and will not, go ahead if there is a chance of being pruned by a cut with a smaller scope. The third type of pruning operator is the *cavalier commit* which is executed immediately, even if endangered by a smaller cut. The cavalier commit is provided for experimental purposes only, it is expected to be used in exceptional circumstances, for operations similar to `abort` in Prolog. Work done in the scope of a pruning operator is said to be *speculative*.

Suspension is used to preserve the observable semantics of Prolog programs executed by Aurora: when a built-in predicate with some side-effect is reached on a non-leftmost branch of the search tree, or when a pruning operator is reached on a branch which could be pruned by a cut with a smaller scope, the execution must be suspended. Furthermore the scheduler may decide to suspend the current branch when less speculative work can be done somewhere else in the tree.

Four separate schedulers are currently being developed for Aurora. The Argonne scheduler [6] relies on data stored in the tree itself to implement a local strategy according to which live nodes "attract" workers without work. When several workers are idle they will compete to get to a given piece of work and the fastest one will win. The Manchester scheduler [8] tries to select the nearest worker in advance, without moving over the tree. It uses global data structures to store some of the information on available work and workers. The wavefront scheduler [5] uses a special distributed data structure, the *wavefront*, to facilitate allocation of work to workers. The Bristol scheduler [3] tries to minimise scheduler overhead by extending the public region eagerly: sequences of nodes are made public instead of single nodes, and work is taken from the bottommost live node of a branch.

5.3 The Top Level View of the Interface

The principal duty of the scheduler is to provide the engine with work. The thread of control thus alternates between the two components: the engine executes a piece of Prolog code, then the scheduler finds the next assignment, passes control back to the engine, etc. A possible way of implementing this interaction is to put the scheduler *above* the engine: the scheduler *calls* the engine when it finds a suitable piece of work to be executed and the engine *returns* when such an assignment has been finished. In fact this scheme was the basis of earlier interfaces in Aurora [7].

We use a different approach in the current version of Aurora. The execution is governed by the engine: whenever it finishes an assignment, it calls an appropriate scheduler function to provide a new piece of work. The advantage of this scheme is that the environment for Prolog execution (e.g. the set of WAM-registers) is not destroyed when an assignment is terminated and need not be rebuilt upon returning to work. This is of special importance for Prolog programs with fine granularity (i.e. small assignment size), where switching between engine and scheduler code is very frequent [14].

Figure 5.2 shows the top view of the current interface. This is centered around the engine doing work. All the other boxes in the picture represent scheduler functions called by the engine. Note the convention that the names of all scheduler functions are prefixed with '`Sched_`'.

The functions shown in Figure 5.2 are arranged in three groups:

- finding work (left side of Figure 5.2);
- communication with other workers during work (lower part of Figure 5.2), e.g. when cuts or side effect predicates are to be executed;
- certain events during work that may be of interest to the scheduler (right side of Figure 5.2), e.g. creation and destruction of nodes.

The four boxes on the left of Figure 5.2 represent the so called *functions for finding work*:

`Sched_Start_Work` is used to acquire work for the first time, immediately after the initialisation of the worker;

`Sched_Die_Back` is called when the engine backtracks to a public node;

`Sched_Be_Pruned` is invoked when the worker's current branch is pruned off by another worker;

`Sched_Suspend` is called when the worker has to suspend its current branch.

These functions differ in their initial activities, but normally continue with a common algorithm for "looking for work" (see Section 5.5). This algorithm has two possible outcomes: either work is found, or the whole system is halted. Correspondingly each of the functions for finding work has two exits: the normal one (shown on the right side of the function boxes in Figure 5.2) leads back to work, while the other exit (left hand side) leads to the termination of the whole Aurora invocation.

The next group of interface functions provided by the scheduler is depicted at the bottom of Figure 5.2. These functions are called during work, when the engine may require some assistance from the scheduler (mainly in order to communicate with other workers):

`Sched_Prune` — when a cut or commit is executed;

`Sched_Synch` — when a predicate with side effects is encountered;

`Sched_Check` — at every Prolog procedure call (to check for interrupts).

The above functions have three exits. The normal exit (depicted by upwards arrows in Figure 5.2) leads back to work. The other two exits correspond to premature termination of the current assignment, when the current branch has been pruned or has to suspend (leftward and downward arrows). In both cases the engine will do the housekeeping operations necessary for the given type of assignment termination, and proceed to call the scheduler to find the next assignment. See Section 5.6 for a more detailed description of the functions for communication with other workers.

The third group of functions shown in Figure 5.2 (right hand side) corresponds to some events during work that may be of interest to the scheduler. A common property of this group is that the interface does not prescribe any specific activity to be done by these functions: the scheduler is merely given an opportunity to do whatever is needed for maintaining its data structures. As an example, `Sched_Node_Created` (and the corresponding `Sched_Node_Destroyed`) can be used to keep track of the presence of parallel nodes in the private region—as a prospective source of work for other workers. Similarly `Sched_Clause_Entered` can be utilised for maintaining information about the presence of pruning operators in the current branch (see Section 5.7.2).

There are further groups of scheduler functions, not shown in Figure 5.2. These are used in the initialisation of the whole system, in handling keyboard interrupts , and in the implementation of certain optimisations (Section 5.7.1).

The engine side of the interface consists of several groups of functions that support the scheduler algorithm:

- providing access to certain data structures (nodes and alternatives) maintained by the engine,
- extending the public region on the current branch of execution,
- positioning the engine (i.e. the binding array) in the search tree, while looking for work,
- notifying the engine of certain events, e.g. work being found.

The data structure aspects of the engine interface are presented in Section 5.4. Other interface functions provided by the engine will be described in Sections 5.5 and 5.6.

5.4 Common Data Structures

The engine is responsible for maintaining the *node stack*, a principal data area of major importance to the scheduler. The engine defines the *node* data type, but the scheduler is expected to supply a number of fields to be included in this structure for its own purposes.

Among the node fields defined by the engine, some are of interest to the scheduler. Access functions for these fields are provided in the interface:

`Node_Level` — the distance of the node from the root of the search tree,

`Node_Parent` — a pointer to the parent node in the tree,

`Node_Alternatives` — a pointer to the next unexplored alternative of the node.

The scheduler-specific fields of the node data structure normally include pointers describing the topology of the tree. For example, most schedulers will have fields storing a pointer to the first child and the next sibling of a node.

An additional common static data structure, the *alternative*, is introduced to allow the schedulers to keep static data related to clauses. This data structure is used in the Aurora engine to replace the '`try`', '`retry`' and '`trust`' instructions of WAM [9]. Each clause of the user program is represented by an alternative, which stores a pointer to the code of the clause and a pointer to the successor alternative, if any. If a

predicate is subject to indexing, the compiler may create several chains of alternatives to cater for different values in the indexing argument position. This means that several alternatives can refer to the same clause.

The scheduler may supply a number of fields to be included in the alternative structure, to accommodate any (static) information to be associated with clauses. The scheduler can derive this data from the information supplied by the engine when alternatives are created (`Sched_Alternative_Created`). There are two types of static data supplied by the engine:

- information about sequential predicates—this information is normally stored in each alternative of the predicate.
- pruning information—data on the number of pruning operators (cuts, commits and conditional expressions) contained in the clause or the predicate (see Section 5.7.2).

The only engine field in the alternative structure that is of interest to the scheduler is the one pointing to the successor alternative (`Alternative_Next`). This field is used, for example, when the scheduler starts a new branch from a public node and needs to advance the next alternative pointer of the node.

5.5 Finding Work

Figure 5.3 shows the engine functions used by the scheduler while it is looking for work. The actual algorithms of the four functions for finding work will normally differ, but they all use the same set of engine support functions.

Functions `Move_Engine_Up` and `Move_Engine_Down`, shown on the right hand side of Figure 5.3, instruct the engine to move the binding array up or down the current branch. Initially, the binding array is positioned at or below the youngest public node on the branch. Before returning, the scheduler has to position the binding array above the new sentry node.

Different schedulers employ different strategies in moving over the tree. The Argonne scheduler moves node-by-node, when approaching the potential work node. Other schedulers locate a piece of work from a distance and move the engine to the appropriate place in a few big jumps.

There is no need to move the engine if work is taken from the parent of the old sentry node. An additional entry point to the scheduler, `Sched_Get_Work_At_Parent` (see Section 5.7.1), has been provided for this special case.

The left hand side of Figure 5.3 shows the engine functions for memory management of the node stack. A worker may have to remove some dead nodes from the tree as it moves upwards. This involves deleting these nodes from the scheduler data structures (normally the sibling chain) and invoking the `Mark_Node_Reclaimable` engine function. As a special case, the old sentry node will have to be deleted from the tree at the beginning of `Sched_Die_Back` and `Sched_Be_Pruned`.

When the scheduler decides to reserve a new piece of work from a live public node (work node), it has to create a sentry node for the new branch. This involves calling the `Allocate_Node` function, which first removes all the nodes that have been marked as reclaimable from the top of the worker's stack and then allocates a new sentry node. The related `Allocate_Foreign_Node` function is used if *another* worker allocates a node on the stack of the worker looking for work. This is used in the Manchester scheduler to implement handing work to an idle worker.

The new sentry node serves as a placeholder for the new assignment. The scheduler inserts the sentry into the search tree and simultaneously reserves an alternative to be explored by the new branch (by reading and advancing the `Node_Alternatives` field of the work node).

The bottom part of Figure 5.3 shows the possible exit paths from the functions for finding work. The actual work found can correspond either to a new branch or to a branch which was hitherto suspended and can be resumed now. Functions `Found_New_Work` and `Found_Resumed_Work` are used to notify the engine about the type of the work found, and to supply the new sentry node. The box for `Found_New_Work` in Figure 5.3 shows the `SENTRY` argument to highlight the fact that this argument should be the same as the one returned in `Allocate_...Node`.

5.6 Communication with Other Workers

The need for communication with other workers arises when a pruning operator or a built-in predicate with side effects is to be executed. In addition, a periodic check is needed to examine if there are communication requests from other workers.

The `Sched_Prune` function is invoked when a pruning operator is encountered. At this moment the engine has already executed the private part of the pruning. The scheduler receives a pointer to the cut node (showing the scope of pruning) and an argument indicating the type of the pruning operator (cut, commit or cavalier commit). It has to check if the preconditions for pruning are satisfied: the current branch should not be pruned itself, and, except for the cavalier commit, it should not be endangered by cuts with a smaller scope, as discussed in [10]. The latter condition can be replaced by a requirement for the branch to be leftmost in the subtree rooted at the child of the cut node, if the scheduler does not maintain specific pruning information.

If the preconditions of pruning are not satisfied, `Sched_Prune` uses one of the abnormal exits (cf. Figure 5.2) to indicate that the branch has been killed or that it has to suspend (waiting to become leftmost). If the pruning operation can go ahead, the scheduler has to locate the workers that are in the pruned subtree and interrupt them. There may be branches in this subtree which have previously been suspended. A special engine function, `Mark_Suspended_Branch_Reclaimable`, is used for cleaning up such branches.

The `Sched_Synch` function is invoked when a call to a built-in predicate with side-effects is encountered. Normally such calls are executed only when their branch becomes leftmost in the whole tree. There are, however, some special predicates (e.g. those used to assert solutions in a `setof`), for which the order of invocation is not significant: their execution can go ahead if not endangered by a cut within a specific subtree. The `Sched_Synch` function receives an argument encoding the type of the check needed, and a pointer to the root of the subtree concerned.

The third communication function, `Sched_Check`, is called at every Prolog procedure call. Frequent invocation of this function is necessary so that the scheduler can answer requests (e.g. interrupts) from other workers without too much delay. Note, however, that a scheduler may choose to do the checks only after a certain number of `Sched_Check` invocations (as is the case for the Manchester and Argonne schedulers).

The nature of requests to be handled by `Sched_Check` varies from scheduler to scheduler. There are, however, two common sets of circumstances: the worker may be requested to kill its assignment or to make some of its private nodes public (to make work available to other workers). The latter activity needs assistance from the engine: the function `Make_Public` extends the public region on the current branch down to a specified node.

5.7 Extensions of the Basic Interface

5.7.1 Simplified Backtracking

When a worker backtracks to a live public node and is able to take a new branch from there, several administrative activities can be avoided. The sentry node can be re-used, rather than being marked as reclaimable and re-allocated. There is scope for a related optimisation in the scheduler: instead of deleting the old sentry from the sibling chain and then installing it as the last sibling, the scheduler can move the sentry node to the end of the sibling chain (or do nothing if the old sentry was the last child). The interface supports this important optimisation by a function `Sched_Get_Work_At_Parent`, called when the engine backtracks to a live public node. If the scheduler, following the necessary synchronisation operations, still finds the node to be live, it can reserve an alternative from that node. If the scheduler cannot take work from the node in question, it returns to the engine, which will subsequently invoke `Sched_Die_Back` to acquire a new piece of work.

The `Sched_Get_Work_At_Parent` function also supports the *contraction* operation of the SRI model [17]. This operation removes a dead nonfork node after the last alternative has been taken from it. The node in question can be physically removed only if it is on the top of the stack of the worker executing the given branch.

5.7.2 Pruning Information

Information about the presence of pruning operators in a clause may be needed by the scheduler to perform pruning more efficiently or to distinguish between speculative and non-speculative work. Various algorithms related to pruning have been developed and discussed in [10]. When designing the interface, we tried to generalise and extend the format of pruning data as described in [10], so that other possible approaches (e.g. [13]) can be supported as well.

If one disregards disjunctions, the information needed about pruning is quite simple. A scheduler may wish to know whether a clause contains cuts or commits[3]. For more exact pruning algorithms the number of occurrences of each pruning operator may be needed. The fact that a clause must fail, may also be of interest: when such a clause is entered, the pruning operators in the current continuation (i.e. in the previous resolvent) become inaccessible. The simple set of pruning data would thus consist of three items for each clause: the number of cuts, the number of commits and the Boolean value indicating whether the clause ends in a failing call (i.e. `fail`, but in the future, global compile time analysis might discover this property for other calls).

The presence of disjunctions and conditionals makes the situation more complicated. In [15] we present a set of pruning data consisting of seven items, to describe the pruning properties of a general clause (one that may contain disjunctions and conditionals).

5.8 Implementation of the Interface in the Aurora Engine

The Aurora emulator [9] was produced by modifying the SICStus emulator to support the SRI model and by converting it from a stand-alone program to an Aurora worker component connected by an algorithmic interface to a scheduler component. The total performance degradation resulting from these changes has been found to be around 25%. In an earlier paper [11] we gave an overview of the changes imposed by the SRI model. In this section we concentrate on the impacts of the interface on the engine and on changes introduced in the new design.

5.8.1 Boundaries

The engine needs to maintain the boundary between the public and private regions. Within the private region, it must distinguish between *local* nodes, i.e. nodes adjacent to the top of the worker's own stack, and remote nodes. This is achieved by storing a pointer to the respective boundary nodes in certain registers. These registers are initialised when an assignment is started (`Found`...`Work`). They are updated when the public region is extended (`Make_Public`) or contracted (`Sched_Get_Work_At_Parent`), and when backtracking in the private region winds back to the worker's own stack. They are consulted to distinguish different cases of backtracking and pruning operations.

5.8.2 Backtracking

From the engine's point of view, the main complication of or-parallel execution is its impact on the backtracking routine. This routine has to check whether it is about to backtrack into the public region, in which case the scheduler must be invoked to perform public backtracking (`Sched_Die_Back` or `Sched_Get_Work_At_Parent`). Private backtracking has to face the complication that the private region may extend to other workers' stacks, and possibly wind back to the worker's own back again. As explained earlier, remote nodes cannot be reclaimed when they are trusted; instead, `Mark_Node_Reclaimable` is invoked when dying back over a remote node.

Shallow backtracking is optimised in the private region, but only if the current node is on the top of the worker's own stack.

[3] Note that data on cavalier commits is not included in the pruning information, as this operation is expected to be used only for handling exceptional circumstances.

5.8.3 Memory Management

As stated earlier, the stack memory management relies on the node stack. While finding work, each worker maintains a pointer to the youngest node that has to be kept for the benefit of other workers. Such pointers are used and updated by the `Allocate`...`Node` functions. When an assignment is started (`Found`...`Work`) the top of stack pointers for the other WAM stacks are initialised from relevant fields of the node physically preceding the embryonic node of the new assignment, as these fields define how much of the other stacks has to be kept.

5.8.4 Pruning Operators

Pruning operations must distinguish between (i) pruning local nodes only, (ii) pruning remote nodes, and (iii) pruning public nodes. In cases (i) and (ii), the node can be pruned right away, but the memory occupied by the pruned node can only be reclaimed in case (i). The trail must be tidied in all three cases, as explained in [11]. In case (iii), the scheduler is responsible for pruning the public nodes, but may decide to suspend or abort the current assignment instead, forcing the engine to invoke `Sched_Suspend` or `Sched_Be_Pruned`, respectively. Note that `Sched_Prune` is invoked in all three cases, to give the scheduler an opportunity to keep pruning information up to date.

To support suspension of cuts and commits, the compiler provides extra information about what temporary variables need to be saved until the suspended task is resumed. This extra information also encodes the type of the pruning operator.

5.8.5 Premature Termination

To suspend the current assignment when the scheduler uses the "suspend" exit in `Sched_Prune`, `Sched_Synch`, or `Sched_Check`, the engine creates an auxiliary node which stores the current state of computation and calls `Sched_Suspend`. It is up to the scheduler to decide when the suspended work may be resumed.

To abort the current assignment when the scheduler uses the "be_pruned" exit in the above functions, the engine deinstalls all conditional bindings made by the current assignment, marks all remote nodes as reclaimable except the sentry node, and calls `Sched_Be_Pruned`.

5.8.6 Movement

While executing Prolog code, the binding array is kept in phase with the trail stack: whenever a binding is added to or removed from the trail, the bound value is also stored or erased in the binding array. While finding work, the engine maintains a pointer to a node in the tree corresponding to the current contents of the binding array. When the scheduler asks the engine to "move" the binding array up to a new position (`Move_Engine_Up`), bindings which were recorded on the trail path between the current and the new position are deinstalled from the binding array, and the current position is updated. Similarly, `Move_Engine_Down` installs a number of trailed binding in the binding array and updates the current position.

When an assignment is started (`Found`...`Work`), the engine positions its binding array at the tip node of the new or resumed branch in order to get ready to start executing the Prolog code.

5.9 Applying the Interface to Andorra-I

The engine-scheduler interface has been originally designed for the Aurora or-parallel Prolog system. Its primary purpose has been to support exchangeable use of several schedulers with a single engine (i.e. the Aurora engine based on Sicstus). Recently the interface has been used to link the and-parallel engine of the Andorra-I system with the Bristol scheduler developed in the context of Aurora.

In contrast with the Sicstus engine, Andorra-I performs and-parallel execution: any goals which can be reduced without making choicepoints (so called determinate goals) are executed eagerly in parallel; a team of workers work together to exploit and-parallelism. However, when no determinate goals remain, Andorra-I

behaves similarly to Prolog: it uses the leftmost goal to make a choicepoint. Moreover, the backtracking routine resembles Prolog, as well: when a goal fails, the team backtracks to the nearest choicepoint, and starts to explore the next branch. Thus, despite the and-parallel execution phase, Andorra-I and Aurora behave in exactly the same way in exploring the or-tree. From the point of view of the interface, an Andorra-I team is exactly the same as an Aurora worker.

In the Andorra-I implementation the following data structures have been introduced to support the interface. First, in a way similar to Aurora, Andorra-I requires two additional pointers for each team: one for marking the boundary between the public and the private regions of the tree, and another for storing the current binding array position. Second, a parent pointer has to be added to each node (Andorra-I originally did not require the parent pointer because of the fixed node size). The backtracking routine is modified so that engine always calls the scheduler (`Sched_Die_Back`), if it is in the public region. To simplify the implementation, Andorra-I currently does not allow a worker to work on other workers' stacks. Therefore, when a worker resumes a suspended branch which belongs to someone else, the branch has to be made public.

The main difference between Aurora and Andorra-I arises in the handling of pruning operators. According to the interface, the engine should call the scheduler whenever it executes a pruning operator (`Sched_Prune`). If the scheduler decides that the pruning cannot go ahead, the engine is required to suspend the current branch and call `Sched_Suspend` immediately. In Andorra-I, however, the pruning operator is executed during the and-parallel phase, and there might be some other goals being executed simultaneously by fellow workers in the team. When a worker needs to suspend because of the pruning operator, it has to take care of its team, i.e. inform all other workers to stop and then find new work together. In fact, even if there is only one worker in the team, it is not easy to stop the and-parallel execution phase prematurely, without slowing down the whole execution process. Therefore, we have decided to let the team carry on the and-parallel phase and suspend later, if necessary. As a special case it may happen that the computation fails after `Sched_Prune` is called. In this case, the Andorra-I engine marks the suspended node as a *cut_fail* node. Later on, when the scheduler resumes the given branch, the engine will backtrack immediately.

Preliminary performance results of the Andorra-I system are very promising [2], showing that Andorra-I is capable of exploiting or-parallelism with similar efficiency as in Aurora. The overall experience of using the interface in the Andorra-I implementation is very positive: the interface proved to be well designed and of appropriate abstraction level.

5.10 Performance Results

No detailed performance analysis work has been done for the new Aurora implementation yet. Preliminary measurements have been performed with the Manchester scheduler, on the benchmark suite introduced in the performance analysis of the earlier Aurora version [14]. The benchmarks are divided into three groups according to granularity: course granularity (top section in the tables), medium granularity (middle section), and fine granularity (bottom section).

Table 5.1 shows the running times for that benchmark suite on the first generation of Aurora (using the old interface and an engine based on Sicstus Prolog 0.3). Table 5.2 shows the running times for the same benchmarks in the second generation of Aurora. There is an overall improvement of up to 60% in terms of absolute speed, mostly due to the new, much faster engine. For some of the fine granularity benchmarks the relative speedups have deteriorated; this is because the increase in engine speed implies a relative increase in scheduler overheads. For benchmarks with coarse granularity, and especially for the ones with frequent suspension and resumption (e.g. `tina`), the relative speedups have improved, showing the advantages of the new interface.

5.11 Conclusions and Future Work

We have described the engine-scheduler interface used in the second generation of the Aurora or-parallel Prolog system. We have defined a simple set of functions to cover the two basic areas of engine-scheduler interaction: finding work and communication between workers. We have identified those events during Prolog execution that may be of potential interest to schedulers, e.g. creation of nodes, entering clauses, etc. We have also developed a general characterisation of pruning properties of Prolog clauses that can be used both for scheduling speculative work and for improving the implementation of pruning operators.

Goals * repetitions	Aurora Workers 1	4	8	11	Sicstus 0.3
8-queens1	10.11	2.54*(3.98)*	1.29*(7.84)*	0.97*(10.4)*	8.19*(1.23)*
8-queens2	29.37	7.32*(4.01)*	3.73*(7.87)*	2.76*(10.6)*	23.60*(1.24)*
tina	21.30	5.57*(3.83)*	3.02*(7.06)*	2.37*(8.98)*	17.29*(1.23)*
salt-mustard	11.71	3.03*(3.87)*	1.63*(7.18)*	1.27*(9.24)*	9.50*(1.23)*
AVERAGE		*(3.92)*	*(7.49)*	*(9.80)*	*(1.23)*
parse2 *20	9.24	2.92*(3.17)*	2.08*(4.44)*	1.96*(4.72)*	7.54*(1.23)*
parse4 *5	8.54	2.50*(3.42)*	1.67*(5.11)*	1.40*(6.10)*	6.91*(1.24)*
parse5	6.02	1.74*(3.46)*	1.17*(5.15)*	0.98*(6.14)*	4.89*(1.23)*
db4 *10	3.12	0.87*(3.60)*	0.53*(5.87)*	0.45*(6.96)*	2.69*(1.16)*
db5 *10	3.80	1.04*(3.66)*	0.64*(5.93)*	0.55*(6.92)*	3.28*(1.16)*
house *20	8.13	2.26*(3.60)*	1.40*(5.81)*	1.19*(6.84)*	6.51*(1.25)*
AVERAGE		*(3.48)*	*(5.38)*	*(6.28)*	*(1.21)*
parse1 *20	2.49	0.90*(2.77)*	0.81*(3.08)*	0.87*(2.87)*	2.02*(1.23)*
parse3 *20	2.13	0.84*(2.54)*	0.80*(2.66)*	0.83*(2.57)*	1.72*(1.24)*
farmer *100	4.83	2.34*(2.06)*	2.41*(2.00)*	2.49*(1.94)*	3.80*(1.27)*
AVERAGE		*(2.46)*	*(2.58)*	*(2.46)*	*(1.25)*

Table 5.1: Run times, first generation of Aurora

The interface described in this paper is fundamentally revised with respect to earlier versions. The new interface is designed to help avoid scheduling overheads, to make the set of basic concepts simpler and more uniform, to give scope for potential optimisations including better memory management, improved treatment of pruning operations, and avoidance of speculative work.

The main purpose of the interface is to enable different engines and schedulers to be used interchangeably. To date, four separate schedulers have been written and connected to two different engines by means of the interface. Perhaps more importantly, the evolution of the interface has helped clarify many issues in implementing or-parallelism in Prolog, such as contraction and handling of pruning information.

The interface has contributed to the overall improvement of Aurora performance. We also believe that the new interface has played a significant part in the good performance results of the Bristol scheduler. The Bristol scheduler has been designed with the new interface in mind, and, in spite of applying a very simple scheduling strategy, its performance is comparable (and sometimes better than) that of the earlier schedulers [3].

The main outstanding issue which has not been treated in the interface is garbage collection. Patrick Weemeeuw [18] has addressed the problem of garbage collection of the public parts of the tree. Since such activities involve synchronisation between workers and possibly relocation of scheduler data, it is likely that the interface will have to be extended to support garbage collection.

The interface has recently been utilised in a project based on the Muse approach to or-parallel Prolog [1]. An or-parallel version of BIM_Prolog [4] is currently being produced by modifying the BIM engine and connecting it via the interface to the Muse scheduler.

We are convinced that the applicability of the interface extends beyond or-parallel Prolog systems. The Andorra experience is powerful evidence of this fact, but it must be stressed that in this case, the interface was used to add or-parallelism to an already and-parallel system. Generalising the interface to cover issues of and-or-parallel scheduling could be an interesting research direction to be pursued in the future.

5.12 Acknowledgements

The work on engine-scheduler interfaces was initiated by David Warren. Earlier versions of the interface were developed by Ross Overbeek and Alan Calderwood. The design of the new interface benefited from

Goals * repetitions	Aurora Workers 1	4	8	11	Sicstus 0.6
8-queens1	8.01	2.03*(3.95)*	1.03*(7.75)*	0.76*(10.6)*	6.77*(1.18)*
8-queens2	20.63	5.25*(3.93)*	2.64*(7.81)*	1.93*(10.7)*	16.45*(1.25)*
tina	18.40	4.65*(3.96)*	2.39*(7.69)*	1.79*(10.3)*	13.78*(1.34)*
salt-mustard	10.89	2.82*(3.86)*	1.48*(7.36)*	1.11*(9.86)*	8.85*(1.23)*
AVERAGE		*(3.92)*	*(7.65)*	*(10.4)*	*(1.25)*
parse2 *20	7.16	2.40*(2.99)*	1.71*(4.18)*	1.64*(4.37)*	5.87*(1.22)*
parse4 *5	6.67	1.85*(3.60)*	1.40*(4.76)*	1.19*(5.60)*	5.40*(1.24)*
parse5	4.71	1.42*(3.33)*	0.96*(4.89)*	0.81*(5.81)*	3.82*(1.23)*
db4 *10	2.94	0.81*(3.63)*	0.46*(6.39)*	0.38*(7.82)*	2.24*(1.31)*
db5 *10	3.56	0.97*(3.67)*	0.57*(6.25)*	0.47*(7.64)*	2.73*(1.30)*
house *20	5.07	1.47*(3.46)*	0.93*(5.48)*	0.79*(6.42)*	4.22*(1.20)*
AVERAGE		*(3.45)*	*(5.32)*	*(6.28)*	*(1.25)*
parse1 *20	1.89	0.76*(2.47)*	0.73*(2.61)*	0.78*(2.42)*	1.57*(1.20)*
parse3 *20	1.62	0.72*(2.24)*	0.68*(2.37)*	0.72*(2.25)*	1.34*(1.21)*
farmer *100	3.61	1.92*(1.88)*	2.13*(1.69)*	2.19*(1.65)*	3.06*(1.18)*
AVERAGE		*(2.20)*	*(2.22)*	*(2.11)*	*(1.20)*

Table 5.2: Run times, second generation of Aurora

several discussions with Tony Beaumont, Per Brand, Bogumił Hausman and Ewing Lusk.

The authors are indebted to Feliks Kluźniak, Ewing Lusk, and the anonymous referees for careful reading and valuable comments on drafts of this paper.

This work was supported by ESPRIT projects 2471 ("PEPMA") and 2025 ("EDS").

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse approach to or-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[2] Anthony Beaumont, S. Muthu Raman, Vítor Santos Costa, Péter Szeredi, David H. D. Warren, and Rong Yang. Andorra-I: An implementation of the Basic Andorra Model. Technical Report TR-90-21, University of Bristol, Computer Science Department, September 1990. Presented at the Workshop on Parallel Implementation of Languages for Symbolic Computation, July 1990, University of Oregon.

[3] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible scheduling or-parallelism in Aurora: the Bristol scheduler. In *PARLE 91, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, June 1991.

[4] BIM. BIM_Prolog release 2.4. 3078 Everberg, Belgium, March 1989.

[5] Per Brand. Wavefront scheduling. Internal Report, Gigalips Project, 1988.

[6] 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.

[7] Alan Calderwood. Aurora—description of scheduler interfaces. Internal Report, Gigalips Project, January 1988.

[8] 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.

[9] Mats Carlsson and Péter Szeredi. The Aurora abstract machine and its emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[10] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[11] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[12] Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press, April 1991.

[13] Raed Sindaha. Scheduling speculative work in the Aurora or-parallel Prolog system. Internal Report, Gigalips Project, March 1990.

[14] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[15] Péter Szeredi and Mats Carlsson. The engine-scheduler interface in the Aurora or-parallel Prolog system. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[16] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing engines and schedulers in or-parallel Prolog systems. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, pages 439–453. Springer Verlag, Lecture Notes in Computer Science, Vol 506, June 1991.

[17] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[18] Patrick Weemeeuw. Memory compaction for shared memory multiprocessors, design and specification. In *Proceedings of the North American Conference on Logic Programming.* MIT Press, October 1990.

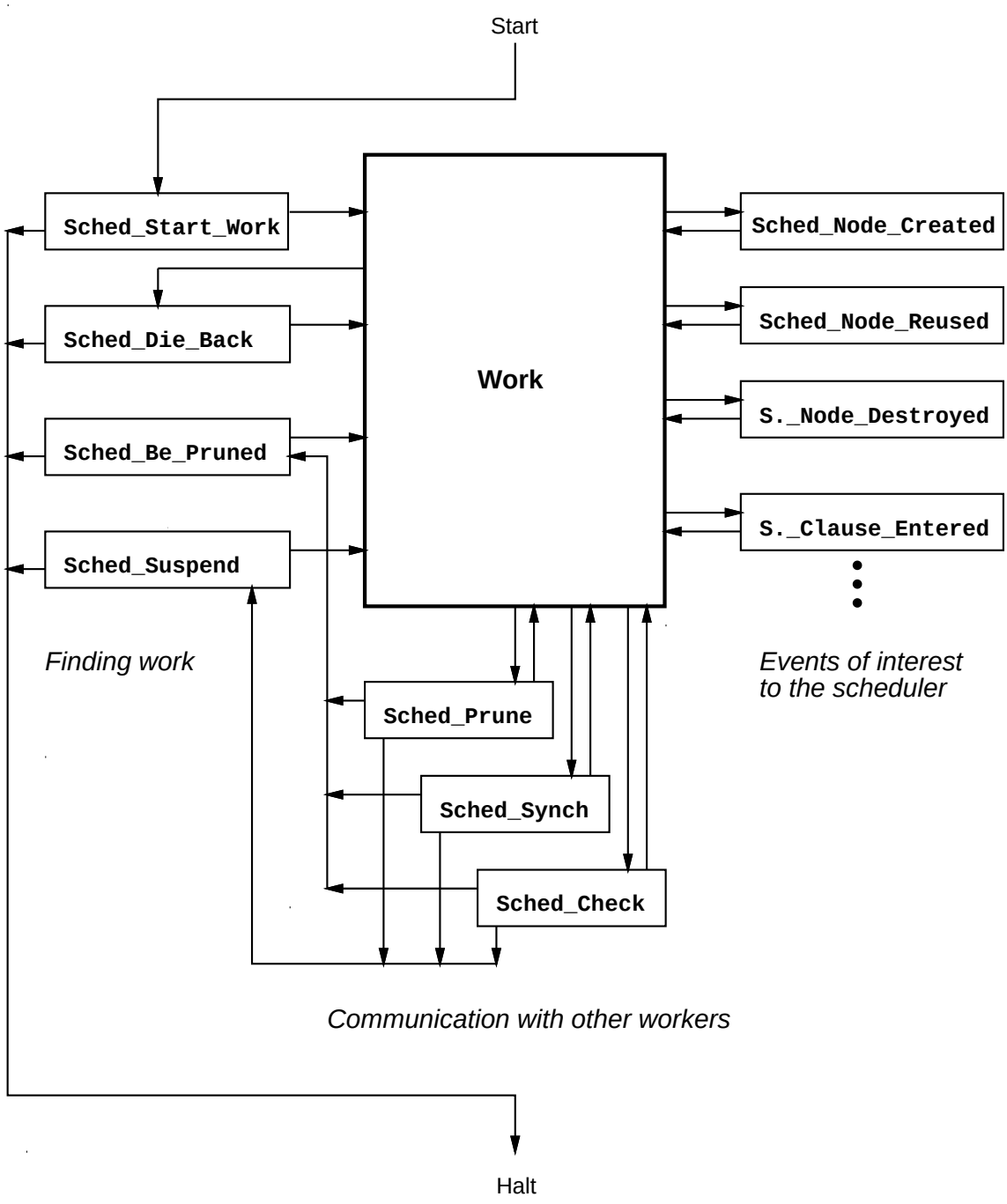

Figure 5.2: The top level view of the interface

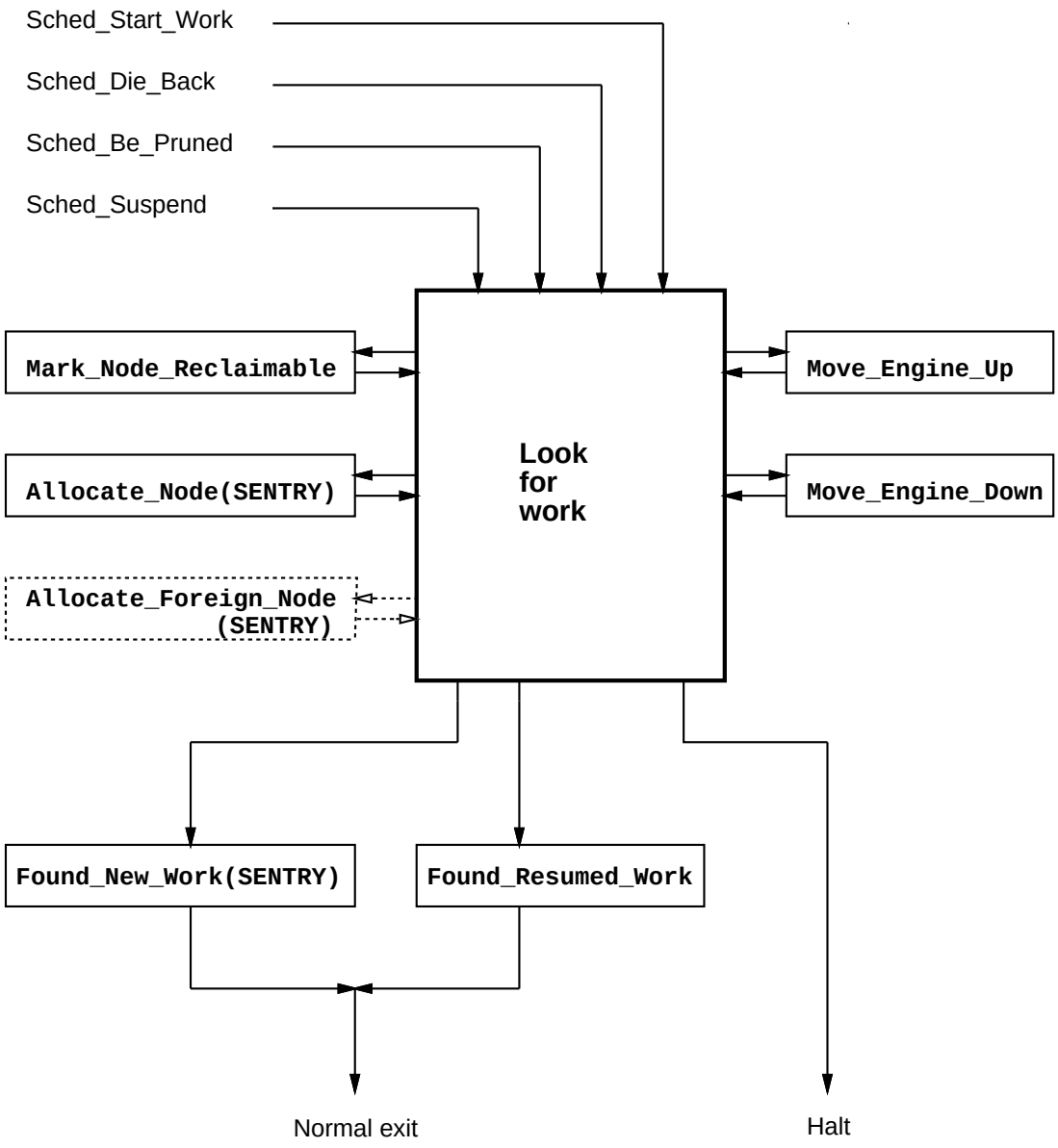

Figure 5.3: Engine functions in looking for work

Part II

Language extensions

Chapter 6

Using Dynamic Predicates in an Or-Parallel Prolog System[1]

Péter Szeredi

SZKI Intelligent Software Ltd. (IQSOFT)
H-1011 Budapest, Iskola u. 10, Hungary
szeredi@iqsoft.hu
and
Department of Computer Science
University of Bristol, Bristol, U.K.

Abstract

Aurora is a prototype or-parallel implementation of Prolog for shared memory multiprocessors. It supports the full Prolog language, thus being able to execute existing Prolog programs without any change. For some programs, however, typically those relying on dynamic database handling, full compatibility with Prolog may cause unnecessary sequencing delays. Aurora therefore supports a number of extensions to Prolog, including asynchronous versions of all side-effect predicates.

Programs often rely on dynamic predicates because of bad programming style. There are, however, applications where dynamic predicates are the most natural way of expressing a solution. In this paper we look at a simple, yet interesting such application: a program for playing the game of mastermind. This is a typical example of a search using a continually changing knowledge base.

We first look at sequential search strategies for playing the game of mastermind. We then proceed to discuss the problems arising when these strategies are executed in parallel, with asynchronous database handling. We present several versions of the mastermind program, discussing various synchronisation techniques and outlining proposals for higher level synchronisation primitives to be incorporated into Aurora. We discuss performance results for the presented programs using an experimental implementation of the synchronisation primitives.

Keywords: Logic Programming, Programming Methodology, Parallel Execution, Synchronisation.

6.1 Introduction

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, currently running on Sequent, Encore and BBN machines. It has been developed in the framework

[1] This paper has appeared in the proceedings of ILPS'91 [14]

of the Gigalips project [7], a collaborative effort between groups at the Argonne National Laboratory in Illinois, the University of Bristol, and the Swedish Institute of Computer Science (SICS) in Stockholm. SZKI Intelligent Software (IQSOFT) in Budapest has recently joined the Gigalips Project.

Aurora uses the SRI model of execution [15]. According to this model the system consists of several *workers* (processes) exploring the search tree of a Prolog program in parallel. Each node of the tree corresponds to a Prolog *choicepoint* with a branch associated with each alternative clause.

Aurora is based on SICStus Prolog, an efficient, portable, Edinburgh style Prolog implementation. A new version of Aurora has been finished recently, which is fairly robust and fully compatible with the underlying SICStus Prolog.

A number of applications have been used to evaluate the efficiency of parallel execution in Aurora [7, 6]. One of the major outstanding problems in this respect is the poor efficiency of programs relying on dynamic predicates.

We would like to emphasise that we do not want to advocate unnecessary usage of dynamic predicates. We believe, however, that there are problems where usage of dynamic predicates is justified. For example there are search problems where the standard depth-first search of Prolog is inadequate: in such cases dynamic predicates can be used for providing the more sophisticated control of the search. Similarly, dynamic predicates may be the most natural way to implement search problems which use a continually changing knowledge base. A simple example of such search problem is the one encountered in a straightforward algorithm for playing the game of mastermind, which is the subject of a case study presented in this paper.

We also believe that the usage of non-declarative features in logic programming languages should be reduced in the future. This can be achieved, for example, by introducing higher order language extensions to encapsulate frequently used algorithms that currently can only be programmed in a non-declarative way. The `bagof` and `setof` predicates of current Prologs are typical examples of such higher order functions. Recently we have developed a proposal for a `maxof` predicate that encapsulates several optimum search techniques, including branch-and-bound and alpha-beta pruning [12]. We plan to develop parallel implementations for such higher order predicates within Aurora itself, in a way similar to how `setof` is implemented in most of the current Prolog systems. We need appropriate synchronisation techniques and tools for this task, thus providing further motivation for the work presented in the sequel.

We first describe the asynchronous dynamic database handling predicates as provided in the present version of Aurora (Section 6.2). We then proceed to the case study of various implementations of the mastermind program. We start with a discussion of basic search strategies for mastermind and presenting a (sequential) Prolog program (Section 6.3) . We then examine how asynchronous built-in predicates can be used to generalise the mastermind search algorithm in a way appropriate for parallel execution. We first deal with the basic synchronisation problems (Section 6.4) and then proceed to discuss several alternative implementations of the mastermind program, suitable for parallel execution (Sections 6.5, 6.6 and 6.7). We present performance results for these program-variants in Section 6.8. We end with a discussion of related work and reiterating the conclusions of the paper. An expanded form of this paper has appeared as [13].

6.2 Extensions to Prolog in Aurora

Aurora supports or-parallel execution of the full Prolog language. To preserve compatibility with Prolog, restrictions in exploiting parallelism have to be introduced in the case of certain non-pure language primitives. The most notable examples of such primitives are the built-in predicates with side effects.

To preserve Prolog semantics, the side-effect predicates have to be executed in exactly the same left-to-right order as in sequential Prolog. In the Aurora implementation this is achieved by suspending the execution of non-leftmost branches that are to invoke a side-effect predicate. Such branches are resumed only when they become leftmost.

Frequent suspension leads to serious overheads and degradation of parallel performance. On the other hand, side-effect predicates are often used in a context where there is no real need to preserve the strict left-to-right order. The present implementation of Aurora therefore provides two additional variants for each side-effect predicate *Pred*. These are written as

`asynch` *Pred* e.g. `asynch assert(foo)` and
`cavalier` *Pred* e.g. `cavalier write(bar)`.

Here `asynch` and `cavalier` are prefix operators. The `asynch` *Pred* call will be executed immediately only if it is not in the scope of a cut operator, to prevent the occurrence of undesired side effects (as discussed by Hausman in [3]). If the `asynch` call is in the scope of a cut, then the current branch will be suspended until the danger of being cut ceases to exist. The cavalier form of the side effect predicate will be executed unconditionally. A typical usage of this form is to display tracing information on how the parallel execution proceeds.

Both the asynchronous and the cavalier predicates are executed atomically. This means that if two competing branches reach a side-effect predicate affecting the same resource[2] simultaneously, then these predicates will be executed in some arbitrary order, one after the other.

Aurora also provides a commit operator, denoted by a vertical bar (|). This is the symmetrical version of the cut operator, which prunes branches both to its left and to its right. The commit is comparable to asynchronous side effect predicates, as it is not allowed to proceed in the scope of a (smaller) cut [3]. Note that Aurora does not provide a completely cavalier commit operator, as this is not a meaningful operation.

6.3 The Game of Mastermind

The game of mastermind is a well known game for two players. One of the players chooses a secret code (normally four pegs of various colours). The other player makes a sequence of guesses until the secret code is found out. For each guess, the first player displays a score, consisting of two types of special scoring pegs: a "bull" is shown for each exact match and a "cow" is given for each inexact match.

There is a natural strategy for playing the game of mastermind, which, in the context of logic programming, was first described by van Emden [2]. The strategy is based on the observation that, at each stage of the game, the scores received so far restrict the set of possible secret codes. Only those secret codes are feasible which are *consistent* with all the guess-score pairs played so far, i.e. which would give identical scores for these guesses. The very simple but effective strategy described by van Emden is the following: determine the next guess as an arbitrary one of the feasible secret codes.

The problem of finding feasible secret codes involves a fairly large search space, hence its suitability for Prolog and for exploiting or-parallelism. There are two basic strategies for exploring this search space, which we will call the multi-search and the single-search approach.

According to the first approach (used in van Emden's original paper), one can view the process of playing the game as consisting of several searches, one for each turn of the game. Each such search is based on the history of previous guesses and corresponding scores, and attempts to find a new guess consistent with this history. As soon as an appropriate guess is found, the search is abandoned, the corresponding score is obtained from the opponent, and the next turn is started with the extended history (unless the game is finished).

An important property of this algorithm for playing mastermind is that each guess has to be tried only once during the whole game[3]. This leads to an alternative, single-search strategy, in which the whole game is implemented as a single scan through the search space of all possible guesses. Every guess is then checked for consistency with the current history, which is updated in each turn of the game. There is only a single pruning operation in this case, when the game is finished. This strategy was used in the variant of mastermind presented by Sterling and Shapiro [11].

There is no significant difference between the efficiency of the two search strategies in a sequential Prolog system. In fact the multi-search strategy can be made to "simulate" the other strategy: if one introduces an arbitrary ordering on the set of all guesses, then each constituent search can start enumerating the guesses from the first yet untried one. If, however, the mastermind search is to be performed in or-parallel, then the drawbacks of the multi-search approach become apparent. Since the search space has to be pruned in each turn, all the effort spent on exploring the search space to the right of the leftmost consistent guess is wasted. It is possible to relax the algorithm, and accept any consistent guess (rather than the leftmost one) for the next turn. In this case, however, there is no straightforward way of excluding the guesses considered so far, and so the whole search space has to be enumerated in each turn.

We will implement the single-search strategy for playing mastermind using the dynamic database of Prolog

[2] e.g. the same dynamic predicate, or the same output stream

[3] This is because any guess which has been considered earlier, has either been found inconsistent with a sub-history, or has been played in a turn and a non-final score has been received.

to store the current history. We will use this program as a vehicle for the exploration of the problems involved in parallel execution of Prolog programs relying on dynamic predicates. We would like to emphasise that our principal goal is to present general problems and techniques rather than to offer improvements to the mastermind program itself[4].

```
mastermind(Code):-
   init_history,
   generate_guess(Code),
   current_history(History), consistent_history(Code, History),
   ask(Code, Score), extend_history(Code, Score),
   final_score(Score), !.

% Return the current history.
current_history(H):-
   history(H).

% Add the Code-Score pair to the front of the current history.
extend_history(Code, Score):-
   retract(history(H)), assert(history([Code-Score|H])).

% Set up an empty history.
init_history:-
   retractall(history(_)), assert(history([])).

% consistent_history(Guess, History)
% Guess is consistent with each code-score pair in the History list.

% generate_guess(Guess)
% Guess is a guess.

% ask(Guess, Score)
% Score is the score for Guess.

% final_score(Score)
% Score is final, i.e.  the secret code has been found out.
```

Figure 6.1: The Prolog program for the game of mastermind

Figure 6.1 shows a version of the mastermind program based on the single search approach. It is similar to the one presented by Sterling and Shapiro, but the whole history list is stored in a single clause, rather then having a separate clause for each guess-score pair. This makes our initial discussion of parallel execution simpler—we will consider a multi-clause representation in Section 6.6.

The program is based on a generate and test loop. We start with initialising our data structure: asserting an empty list as the current history (`init_history`). We then proceed to enumerate all possible secret codes (`generate_guess`), and check their consistency against the current history as retrieved from the database (`current_history`[5]). If a code is found to be consistent, we proceed to make a turn, i.e. ask for the corresponding score and extend the history accordingly (`extend_history`). We now check if the score received is final, in which case the remaining choices are pruned and the game is finished. Otherwise we backtrack to `generate_guess` and continue checking the remaining codes.

Figure 6.1 also shows a brief description of the predicates implementing the lower layer of the mastermind program. This lower layer, which is common to all examples of this paper, is presented in full in [13].

6.4 Synchronisation Primitives in Aurora

The mastermind program of Figure 6.1 is inherently sequential. When run on the Aurora or-parallel Prolog system, all the predicates that access or modify the dynamic database are executed in strict left-to-right order. This means that only the call to `generate_guess` will be expanded in parallel. As soon as execution

[4] As van Hentenryck [5] pointed out, constraint based techniques yield very good results in improving the efficiency of the mastermind algorithm.

[5] The `current_history` predicate is introduced here for the sake of future, more complex, versions of the program.

reaches the call of `history` in `current_history`, the branch has to suspend and wait until it becomes leftmost.

As discussed in Section 6.2, Aurora provides an asynchronous version for each dynamic database handling predicate. It is fairly obvious, however, that replacing all dynamic database predicates by their asynchronous counterparts does not produce the desired effect. What we need in the first place is the atomicity of more complex operations, such as replacing a clause in a dynamic predicate (e.g. in `extend_history`). In algorithmic languages such complex atomic operations are normally constructed with the help of *locks*. One would thus be tempted to advocate the introduction of locking primitives into Aurora. It may be interesting to note, however, that the presence of the atomic retract operation in Aurora can be used to provide an experimental implementation of locks (Figure 6.2).

```
init_lock(LockName):-
   asynch retractall(lock_data(LockName)),
   asynch assert(lock_data(LockName)).

lock(LockName):-
   repeat, asynch retract(lock_data(LockName)), !.

unlock(LockName):-
   asynch assert(lock_data(LockName)).
```

Figure 6.2: Basic operations on locks

Locks are implemented here using a dynamic predicate `lock_data`. Creating a lock (`init_lock`) means asserting a clause `lock_data(LockName)`, where `LockName` is an arbitrary ground term. The presence of such a clause in the database corresponds to the lock being free. Grabbing the lock (`lock`) is done by retracting the appropriate clause. The repeat-loop ensures that the retract operation is re-tried if the lock is being currently held, thus forming a busy-waiting loop. Freeing the lock is done by asserting the clause again (`unlock`).

One can use such locking primitives to ensure atomicity and exclusivity of dynamic predicate operations (`current_history` and `extend_history` in our case). We believe, however, that these primitives are still too low-level and unsafe in the context of an or-parallel Prolog. Our main concern is that each locking operation should be paired with a corresponding unlock operation. The control structure of Prolog is much richer than that of the traditional algorithmic languages: the execution of a predicate can abruptly end because of failure or because of its being pruned. For the locking scheme to be safe, the programmer thus has to cater for the possibility of both failure and pruning, in all regions of code where locks are held. A further problem can be caused by the fact that, in a system like Aurora, side-effect predicates can cause branches to suspend. The interaction of locking and suspension can lead to undesired behaviour, including deadlocks.

In order to overcome the problems of the locking scheme, we propose a higher level synchronisation primitive, very similar to the notion of *monitor*, as advocated by Boyle et al. [1] in the context of the C language.

The essence of our proposal is that the user should explicitly mark the critical sections of code (i.e. those requiring exclusive access to some resource), by encapsulating such sections in a call

```
asynch_section(Key, Goal).
```

The first argument of the `asynch_section` predicate identifies the type of the critical section, while the second argument is a goal, or a sequence of goals, actually constituting the critical section. The meaning of this construct is to execute `Goal` in such a way that no other `asynch_section(Key,...)` goal (with the same `Key`) will be run during the execution of `Goal`. Since we want to keep the notion of critical section as simple as possible, we assume that `Goal` is determinate[6].

The `Key` argument of `asynch_section` allows several independent critical sections to be established within the same program. It is very similar to the name of a lock, in fact an experimental implementation of `asynch_section` can be easily built using locks, as shown in Figure 6.3.

[6]Note that it is fairly easy to modify the implementation of the `asynch_section` predicate in Figure 6.3 to cater for nondeterministic goals.

```
asynch_section(Key, Goal):-
   lock(Key), asynch_call(Goal), !, unlock(Key).
asynch_section(Key, _):-
   unlock(Key), fail.

asynch_call((Goal, Goals)):-
   !, asynch Goal, asynch_call(Goals).
asynch_call(Goal):-
   asynch Goal.

init_asynch_section(Key):-
   init_lock(Key).
```

Figure 6.3: An experimental implementation of `asynch_section`

Only non-synchronised side-effect predicates are allowed in critical sections. As seen in Figure 6.3, the subgoals textually present in the second argument of `asynch_section` are automatically executed in asynchronous mode. If a side-effect predicate is called indirectly within `asynch_section`, then it has to be explicitly prefixed with `asynch` or `cavalier`. This restriction enables us to avoid suspension within critical sections: if the implementation ensures (possibly via suspension) that the whole `asynch_section` call is embarked upon only when not endangered by cuts, then any non-synchronised side-effect predicate within the `asynch_section` call will be able to proceed without suspension.

The experimental implementation takes care of unlocking (i.e. exiting the critical section) at both the success and the failure exit of `Goal`. We intend to build a lower level implementation of `asynch_section` which will handle the case of the critical section being subject to pruning as well. Such a lower level implementation will also be able to replace busy waiting by suspension, using appropriate scheduling techniques.

6.5 The Parallel Mastermind Program

Let us now turn to producing a parallel version of the mastermind algorithm of Figure 6.1. As a first attempt one could suggest enclosing the predicates that access or modify the `history` dynamic predicate in `asynch_section` calls. This modification, however, is not sufficient, as it does not cater for the interaction of the simultaneous extensions. For example, it may happen that two guesses, both consistent with the current history, are checked simultaneously and then added to the history. Now the second of these extensions may not be consistent with the *whole* current history, as it has not been checked against the most recent extension. This is contrary to the basic assumption of our mastermind algorithm, i.e. that only such guesses are made that are consistent with the history of the game so far.

A simple solution would be to make the asynchronous section bigger so that it covers all operations from accessing the current history up to the possible update of history. This would mean, however, that only the `generate_guess` call would be explored in parallel, and all the remaining calls, being in a critical section, would be sequentialised. We are therefore looking for a solution where significant parts of the computation are done outside the critical section.

```
mastermind(Code):-
   init_asynch_section(mm), init_history,
   generate_guess(Code),
   current_history(History), consistent_history(Code, History),
   asynch_section(mm, (
     history(NewHistory), append(Unchecked, History, NewHistory),
     consistent_history(Code, Unchecked), ask(Code, Score),
     asserta(history([Code-Score|NewHistory])),
      retract(history(NewHistory))
                 )),
   final_score(Score), !.
```

Figure 6.4: The `mastermind` predicate with proper consistency check

Our first attempt to achieve this goal is shown in Figure 6.4. The critical section now contains the following: reading the current state of history again (`NewHistory`), using `append` to calculate the (possibly empty) difference between this `NewHistory` and the portion of history already checked for consistency, and checking the difference for consistency. If the code in question is found to be consistent with the unchecked part of the history then the score is asked for and the history is extended. Note that the cut in the `mastermind` predicate has been replaced by a commit (`|`), so that we do not insist on finding the leftmost solution (as we actually know that the solution is unique).

The repeated consistency check can be moved out of the critical section at the expense of making the whole algorithm slightly more complex. Figure 6.5 shows the final version of the mastermind program using the single clause data representation. A new, recursive procedure

```
ask_if_consistent(Code, CheckedHistory, Score)
```

is introduced with the overall task of checking `Code` against the current history, under the assumption that it has already been checked against `CheckedHistory`.

```
mastermind(Code):-
   init_asynch_section(mm), init_history,
   generate_guess(Code),
   current_history(History), consistent_history(Code, History),
   ask_if_consistent(Code, History, Score), final_score(Score), |.

% Assume Code has already been found consistent with Checked.
% If Code is not consistent with the current history then fail,
% otherwise ask for the Score and extend the history.
ask_if_consistent(Code, Checked, Score):-
   asynch_section(mm, (
     history(Checked),        % Has the whole history been checked?
     ask(Code, Score),
     asserta(history([Code-Score|Checked])),
     retract(history(Checked))
                 )), !.
ask_if_consistent(Code, Checked, Score):-
   current_history(NewHistory),
   append(UnChecked, Checked, NewHistory),
   consistent_history(Code, UnChecked),
   ask_if_consistent(Code, NewHistory, Score).

current_history(H):-
   asynch history(H), !.

init_history:-
   retractall(history(_)), assert(history([])).
```

Figure 6.5: Parallel mastermind with single clause representation

The first clause of `ask_if_consistent` succeeds only if there were no changes in the current history since it has been last read and checked. If this is the case, the opponent is asked for the score and the history extended. The whole body of this clause is a critical section, so that no further changes can take place until the given guess is processed completely.

The second clause of `ask_if_consistent` reads the current history, calculates the difference, checks it, and recursively calls itself. These activities do not have to be included in a critical section.

Both versions of the mastermind program introduced in this section (Figures 6.4 and 6.5) properly implement the mastermind algorithm and at the same time allow the exploration of the search tree to be performed in parallel. Note also that both programs contain critical sections that can fail. For example, in the first clause of `ask_if_consistent` in Figure 6.5, the critical section fails when the current history has been modified since the last check. If we had to program this algorithm using locks, we would have to use a much more complex control structure.

6.6 Using Multiple Clause Data Representation

The synchronisation scheme used in the previous section interferes with the logic of our algorithm, as it requires an extra level of recursion[7] to process the history. This section presents an alternative solution, which avoids this by storing the history as a sequence of clauses and using a single loop for processing the clauses. The clauses are of the form `history(N, Guess-Score)` and express the fact that the guess `Guess` was put forward and the score `Score` received in the `N`th turn.

```
mastermind(Code):-
   init_asynch_section(mm), init_history,
   generate_guess(Code),
   ask_if_consistent(1, Code, Score), final_score(Score), !.

% Assume Code has already been found consistent with all turns
% before Turn.  Fail if turn Turn exists and Code is not
% consistent with it.  If Turn does not exist, make it.
ask_if_consistent(Turn, Code, Score):-
   asynch history(Turn, GuessScore), !,
   consistent_guess(Code, GuessScore),
   Next is Turn+1, ask_if_consistent(Next, Code, Score).
ask_if_consistent(Turn, Code, Score):-
   asynch_section(mm, (
     \+ asynch history(Turn,_),
     ask(Code, Score), asserta(history(Turn, Code-Score))
                 )), !.
ask_if_consistent(Turn, Code, Score):-
   ask_if_consistent(Turn, Code, Score).

init_history:- retractall(history(_, _)).

% consistent_guess(Guess, CodeScore)
% Guess is consistent with the CodeScore pair.
```

Figure 6.6: Parallel mastermind (multiple clause representation)

The program is shown in Figure 6.6. The first argument of the predicate `ask_if_consistent` (`Turn`) is now a number, the serial number of the next turn to be checked for consistency. The first clause of this predicate is applicable when such a turn has already been made, i.e. there is a corresponding `history` clause. In this case the code is checked for consistency against the guess-score pair stored in the history (`consistent_guess(Code, GuessScore)`). When all turns made so far have been found consistent, the first clause of `ask_if_consistent` fails and the second clause is executed. The task of the second clause is to make the next turn, by asking the opponent for the score and extending the history. The whole of this clause is a critical section, which actually contains a repeated check whether `Turn` is still non-existent (`\+ asynch history(Turn,_)`). If the history has been extended by another worker in the meantime, the second clause fails and the third clause causes the predicate to be called again, so that the recent extension is checked for consistency as well. We could have avoided this repetition by transposing the first two clauses. This would mean, however, loss of efficiency as the much less frequently succeeding clause (which also contains a critical section) would be tried first.

6.7 Predicates for Handling Shared Data

Our last version of `mastermind` was based on incremental construction of the `history` predicate. If we view this dynamic predicate as the representation of the history list, we could consider the process of adding clauses to it analogous to the extension of a shared open-ended list. With this view in mind, we propose a set of primitives for incremental building of general Prolog terms in the shared database. The shared terms are referred to by some special data objects, called references. The basic operations on these objects include

[7] in the `ask_if_consistent` predicate—the lower level of recursion is in the predicate `consistent_history` which scans the history list.

the following:

`create_ref(Ref)` creates a new reference to a shared data object in `Ref`.

`expand_ref(Ref, Term, Goal)` associates `Term` with reference `Ref`, having previously executed `Goal`, if this is possible. If `Ref` has already been associated with a value, or `Goal` fails, `expand_ref` fails as well. The whole operation is atomic.

`access_ref(Ref, Term)` returns the `Term` associated with `Ref`, if there is one; otherwise it fails.

```
mastermind(Code):-
   create_ref(History), generate_guess(Code),
   ask_if_consistent(History, Code, Score), final_score(Score), |.

% Check Code for consistency with History.
% If found consistent, make the next turn and return Score.
ask_if_consistent(History, Code, Score):-
   access_ref(History, [GuessScore|RestHistory]), !,
   consistent_guess(Code, GuessScore),
   ask_if_consistent(RestHistory, Code, Score).
ask_if_consistent(History, Code, Score):-
   expand_ref(History, [Code-Score|NewHistory],
      (ask(Code, Score), create_ref(NewHistory))
            ), !.
ask_if_consistent(History, Code, Score):-
   ask_if_consistent(History, Code, Score).
```

Figure 6.7: The mastermind program using references

Figure 6.7 shows the mastermind algorithm based on the above predicates. The structure of the program closely resembles our previous version. The first argument of `ask_if_consistent` is now a reference to the yet unchecked part of the history list. If this reference is instantiated (`access_ref`), then the appropriate consistency check is made, and the recursion continues with the tail of the list. If the unchecked part is empty, an attempt is made to extend the history list, using the atomic `expand_ref` predicate. If this fails, due to the history having been extended in the meantime by another worker, the third clause provides for the repetition of the whole predicate.

Figure 6.8 shows an experimental implementation of the reference-handling primitives. The references are represented by facts of form `ref(Ref, Term)`. Such a fact is added to the database when `Ref` becomes associated with `Term`. In this implementation references are just numbers. Consecutive reference numbers are generated using a counter `last_ref` (cf. `create_ref`). Expanding a reference is a critical section, which first checks if the reference is still unexpanded, then calls the goal argument of `expand_ref`, and finally creates the new reference by asserting it. Accessing a reference simply translates to checking whether the given reference is present in the database. Note that, due to lack of space, the code for initialisation of reference-handling operations is not shown here.

```
create_ref(Ref):-
   asynch_section(last_ref, (
     retract(last_ref(Last)), Ref is Last+1, assert(last_ref(Ref))
                )).

expand_ref(Ref, Term, Goal):-
   asynch_section(ref, (
     \+ asynch ref(Ref, _), Goal, asserta(ref(Ref, Term))
                )).

access_ref(Ref, Term):-
   asynch ref(Ref, Term).
```

Figure 6.8: An experimental implementation of reference-handling primitives

We envisage a lower level implementation of these primitives, in which the references will be actual pointers to shared Prolog terms. Accessing a reference in such an implementation will be a constant time operation.

6.8 Experimental Performance Results

Table 6.1 shows preliminary performance results (on a Sequent Symmetry with 12 processors) for the four versions of the mastermind program presented in this paper: the Prolog version (Figure 6.1), the one using a single clause for storing the history (Figure 6.5), the one with the multiple clause representation (Figure 6.6) and the one using references (Figure 6.7). The experimental implementation of synchronisation predicates (locking, `asynch_section` and reference handling) was used as shown in Figures 6.2, 6.3 and 6.8.

The first column in the table gives the average execution time for the one-worker case while the remaining nine columns show the speedups relative to the first column. Each of the programs was run with three secret codes taken from different parts of the search tree, and measurements for each secret code were repeated three times. The table shows average run times and speedups.

Version	Workers									
	1	2	3	4	5	6	7	8	9	10
	(Time)	(Speedup)								
Prolog	2.18s	0.83	0.83	0.82	0.83	0.83	0.83	0.83	0.83	0.83
single-clause	3.05s	1.69	3.17	3.97	4.13	5.81	6.23	6.46	6.94	7.01
multi-clause	5.56s	2.02	3.63	4.27	4.72	7.24	7.25	8.42	8.51	8.39
reference	5.73s	2.01	3.45	4.30	4.78	6.91	7.12	7.38	8.31	8.32

Table 6.1: Execution times and speedups for the mastermind program

There is a significant increase of the single worker execution time for the parallel versions of mastermind with respect to the Prolog version (see the first column of the table). This is clearly due to the overheads associated with the experimental implementation of synchronisation predicates. On the other hand, the Prolog version shows a constant slow-down of 17–18% when run with multiple workers, due to the overheads of suspension (necessitated by the usage of synchronous database predicates). The other three asynchronous variants show fairly good, sometimes superlinear speedups[8]. These program variants, run with 10 workers, are 3–5 times faster, in terms of absolute speed, then the Prolog version run with a single worker. We believe that these results are very promising.

6.9 Related Work

Work on using side-effect predicates and pruning operators in the context of Aurora was started by Hausman, Ciepielewski and Calderwood [4]. Hausman's thesis [3] contains a detailed discussion of the implementation issues of side-effect predicates. Sehr [10] presents an alternative approach to the implementation of dynamic database predicates, based on incremental updating of the search tree when changes are made in the database. Neither of these papers deals with the issues of explicit synchronisation and atomicity of more complex side-effect operations.

Reynolds and Kefalas [8] addresses the problems of or-parallel execution of search problems in Prolog within their Brave system. They introduce a special database for storing partial results or *lemmas*, with a restricted set of update operators. While this approach is certainly useful for some applications, it is not capable of handling more complex programs, e.g. the parallel versions of mastermind described here. On the other hand, it may be of interest to implement the lemma-handling primitives of Brave using the synchronisation techniques presented in this paper.

Saraswat [9] develops a family of concurrent constraint programming languages, and deals with the issues of synchronisation and atomicity with respect to and-parallel execution. We believe that some of the techniques introduced in the context of (and-parallel) concurrent logic programming can be utilised in or-parallel systems

[8] The superlinearity is due to the decrease in the number of Prolog reduction steps required to find a solution, because of the concurrent exploration of the search space.

as well. In fact, the predicates for creating and accessing shared data (Section 6.7) were inspired by the Ask and Tell constraints as described by Saraswat[9].

6.10 Conclusions and Further Work

We have presented a case study of various programs for playing the game of mastermind in the context of the Aurora or-parallel Prolog system. We have discussed the main problems associated with parallel execution of programs using dynamic database handling predicates. We have shown how asynchronous database handling predicates can be used to generalise the mastermind search algorithm, to make parallel execution more efficient. We have discussed synchronisation techniques in general and presented proposals for two types of higher level synchronisation primitives to be incorporated into Aurora: the `asynch_section` predicate for marking critical sections of database updates, and the set of predicates for incremental construction of shared data, based on the notion of reference. We have shown promising performance results using an experimental implementation of the proposed predicates.

Work on parallel execution of Prolog programs relying on dynamic predicates can be pursued further in several directions. First, a broad spectrum of existing Prolog applications should be examined and "ported" to Aurora. By the term "porting" here we mean an appropriate transformation of the program that eliminates unnecessary sequentialisation. We believe that the synchronisation primitives introduced in this paper, as opposed to the raw atomic asynchronous operations of Aurora, can serve as useful tools in this process.

As a second direction of further work we should mention the problems of interaction between synchronisation predicates and pruning operators. Some progress in this area, extending the work presented in this paper, has been reported in [12].

As already mentioned, our longer term goals include developing higher order predicates that encapsulate some of the algorithms currently requiring dynamic predicates, and at the same time allow efficient parallel execution. The synchronisation techniques and tools presented in this paper have already been used in the implementation of such a higher order predicate (the `maxof` predicate, [12]). We believe that such synchronisation tools are indispensable in further experiments aiming at the development of new higher order predicates of this kind.

6.11 Acknowledgements

The author would like to thank his colleagues in the Gigalips project at Argonne National Laboratory, the University of Bristol, the Swedish Institute of Computer Science and IQSOFT. Special thanks are due to David H. D. Warren for continous encouragement and help in this work, as well as to Mats Carlsson and Feliks Kluźniak for detailed comments on earlier drafts of this paper.

This work was supported by the ESPRIT project 2025 "EDS", the Hungarian National Committee for Technical Development under project G1-11-034, and the Hungarian-U.S. Science and Technology Joint Fund in cooperation with the Hungarian National Committee for Technical Development and the U.S. Department of Energy under project J.F. No. 031/90.

References

[1] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, 1987.

[2] Maarten H. van Emden. Relational programming illustrated by a program for the game of mastermind. Technical Report CS-78-48, Department of Computer Science, University of Waterloo, Ontario, Canada, 1978.

[3] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[9] I am indebted to Vijay Saraswat for enlightening discussions on this topic.

[4] Bogumił Hausman, Andrzej Ciepielewski, and Alan Calderwood. Cut and side-effects in or-parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[5] Pascal van Hentenryck. *Constraint Satisfation in Logic programming*. The MIT Press, 1989.

[6] Feliks Kluźniak. Developing applications for Aurora. Technical Report TR-90-17, University of Bristol, Computer Science Department, August 1990.

[7] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[8] T. J. Reynold and P. Kefalas. OR-parallel Prolog and search problems in AI applications. In *Logic Programming: Proceedings of the Seventh International Conference*, pages 340–354. MIT Press, 1990.

[9] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989.

[10] David C. Sehr. Or-parallel execution of Prolog programs with side-effects. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[11] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1986.

[12] Péter Szeredi. Design and implementation of Prolog language extensions for or-parallel systems. Technical Report, SZKI IQSOFT and University of Bristol, December 1990.

[13] Péter Szeredi. Using dynamic predicates in Aurora – a case study. Technical Report TR-90-23, University of Bristol, November 1990.

[14] Péter Szeredi. Using dynamic predicates in an or-parallel Prolog system. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Logic Programming Symposium*, pages 355–371. The MIT Press, October 1991.

[15] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

Chapter 7

Exploiting Or-parallelism in Optimisation Problems[1]

Péter Szeredi[2]

IQSOFT—SZKI Intelligent Software Ltd.,
Iskola u. 10, H-1011 Budapest, Hungary,
szeredi@iqsoft.hu

Abstract

Several successful multiprocessor implementations of Prolog have been developed in recent years, with the aim of exploiting various forms of parallelism within the Prolog language. Or-parallel implementations, such as Aurora or Muse were among the first to support the full Prolog language, thus being able to execute existing Prolog programs without any change. There are, however, several application areas where the simple built-in control of Prolog execution hinders efficient exploitation of or-parallelism.

In this paper we discuss the area of optimisation problems, a typical application area of this kind. The efficiency of an optimum search can be dramatically improved by replacing the exhaustive depth-first search of Prolog by more sophisticated control, e.g. the branch-and-bound algorithm or the minimax algorithm with alpha-beta pruning. We develop a generalised optimum search algorithm, covering both the branch-and-bound and the minimax approach, which can be executed efficiently on an or-parallel Prolog system. We define appropriate language extensions for Prolog—in the form of new higher order predicates—to provide a user interface for the general optimum search, describe our experimental implementation within the Aurora system, and present example application schemes.

Keywords: Logic Programming, Programming Methodology, Parallel Execution, Optimum Search.

7.1 Introduction

Development of parallel Prolog systems for multiprocessor architectures has been one of the new research directions of the recent years. Implementation techniques have been developed for various parallel execution models and for various types of parallelism. Or-parallel execution models were among the first to be implemented. Several such systems have been completed recently, such as PEPSys [6], Aurora [8], ROPM [9] and Muse [2].

[1] This paper has appeared in the proceedings of JICSLP'92 [16]

[2] Part of the work reported here has been carried out while the author was at the Department of Computer Science, University of Bristol, U.K.

Our present work is based on Aurora, a prototype or-parallel implementation of Prolog for shared memory multiprocessors. Aurora provides support for the full Prolog language, contains graphics tracing facilities, and gives a choice of several scheduling algorithms [4, 5, 3].

One of the major outstanding problems in the context of parallel execution of Prolog is the question of non-declarative language primitives. These primitives, e.g. the built in predicates for modification of the internal data base, are quite often used in large applications. As these predicates involve side effects, they are normally executed in strict left-to-right order. The basic reason for this is the need to preserve the sequential semantics, i.e. compatibility with the sequential Prolog. Such restrictions on the execution order, however, involve significant overheads and consequent degradation of parallel performance.

There are two main directions for the investigation of this problem. First, one can look at using the unrestricted, "cavalier" versions of the side effect predicates. This opens up a whole range of new problems: from the question of synchronisation of possibly interfering side effects, to the ultimate issue of ensuring that the parallel execution produces the required answers. Since one is using the non-logical features of Prolog here, it is natural that the problems encountered are similar to those of imperative parallel languages. We have explored some of these issues in [15].

Another approach, that can be taken, is to investigate why these non-logical features are used in the first place. One can try to identify typical subproblems which normally require dynamic data base handling in Prolog. Having done this, one can then define appropriate higher order language extensions to encapsulate the given subproblem and thus avoid the need for explicit use of such non-logical predicates. A typical example already present in the standard Prolog is the 'setof' predicate: this built-in predicate collects all solutions of a subgoal, a task which otherwise could only be done using dynamic data base handling.

In this paper we attempt to pursue the second path of action for the application area of optimum search problems. Efficient optimum search techniques, such as the branch-and-bound algorithm and the minimax algorithm with alpha-beta pruning, require sophisticated communication between branches of the search tree. Rather than to rely on dynamic data base handling to solve this problem, we propose the introduction of appropriate higher order predicates. We develop a general optimum search algorithm to be used in the implementation of these higher order predicates, which covers both the branch-and-bound and the minimax algorithm, and which can be executed efficiently on an or-parallel Prolog system such as Aurora.

The structure of the paper is the following. Section 7.2 introduces the *abstract domain*, i.e. the abstract search tree with appropriate annotations, suitable for describing the general optimum search technique. Section 7.3 presents our *parallel algorithm* for optimum search, within this abstract framework. Section 7.4 describes appropriate *language extensions* for Prolog, in the form of new built-in predicates, for embedding the algorithm within a parallel Prolog system. Section 7.5 outlines our experimental Aurora *implementation* of the language extensions using the parallel algorithm. In Section 7.6 we describe two *application schemes* based on the language extensions, preliminary *performance data* for which is given in Section 7.7. Section 7.8 discusses related work, while Section 7.9 summarises the conclusions.

7.2 The Abstract Domain

The abstract representation of the optimum search space is a tree with certain annotations. Leaf nodes have either a numeric value associated with them, or are marked as failure nodes. The root node and certain other non-leaf nodes are called *optimum nodes*. These nodes are annotated with either a *min* or a *max* symbol, indicating that the minimal (maximal) value of the given subtree should be calculated. Some non-leaf nodes can be annotated with constraints of form <*relational-op*> *Limit*, where *Limit* is a number, and <*relational-op*> is one of the comparison operators $<$, $\leq$, $>$ or $\geq$. Constraints express some domain related knowledge about values associated with nodes, as explained below. Figure 7.1 shows an example of an annotated tree.

We will use the term *value node* for the non-failure leaf nodes and the optimum nodes together. We define a value function, which assigns a value to each value node. For a leaf node, the value is the one given as the annotation. For a max (min) node, the value is the maximum (minimum) of the values of all the value nodes directly below the given node. If there are no value nodes below an optimum node (i.e. all nodes below are failure nodes), then the value of a max node can be assumed to be $-\infty$ and that of a min node to be $+\infty$. To simplify the initial discussion we will assume that each optimum (and also each constraint node) has at least one value node below it, and so there is no need for infinite values. We will discuss the general case at the end of Section 7.3.

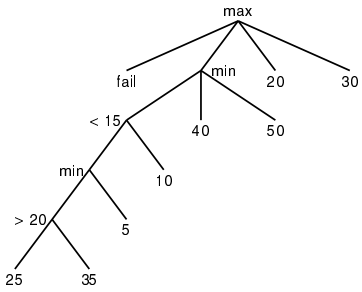

Figure 7.1: An example annotated tree

A node annotated with a constraint <*relational-op*> *Limit* expresses the validity of the following fact:

> For each of the value nodes directly below the constraint node their value V satisfies the following relation: V <*relational-op*> *Limit*.

The example tree in Figure 7.1 contains two constraints. To check that e.g. the upper one (<15) is valid, one has to examine the value nodes directly below (the min node and the leaf node with value 10) both of which do have a value smaller than 15.

The goal of the optimum search is to find the value of the root node. In our example the two min nodes both have a value 5, and the value of the root node is 30.

The notion of search tree presented here is more general than that required by the branch-and-bound and minimax algorithms. The branch-and-bound algorithm uses a search tree with only a single optimum node (the root) and several constraint nodes below. The minimax algorithm applies to trees where there are several layers of alternating optimum nodes but there are no constraint nodes.

We have to introduce a further type of annotation in the search tree to cover some aspects of scheduling: each node can have a numeric priority assigned to it. This priority value will be used to control a best-first type search, i.e. nodes with higher priorities will be searched first (see the examples in Section 7.6).

7.3 The Parallel Algorithm

In our model several processing agents (workers) explore the search tree in parallel, in a way analogous to the SRI model [17]. The workers traverse the tree according to some exhaustive search strategy (e.g. depth-first or best-first) and maintain a "best-so-far" value in each optimum node.

We introduce the notion of *neutral interval*, generalising the alpha and beta values used in the alpha-beta pruning algorithm. A neutral interval, characterised by a constraint of form <*relational-op*> *Limit* can be associated with a particular node if the following condition is satisfied:

> The value of the root node will not be affected if we replace the value of a (value) node directly below the given node, which falls into the neutral interval, by another value falling into the neutral interval.

As the workers traverse the tree they assign neutral intervals to constraint and optimum nodes. When a constraint node is processed, the complement of the constraint interval is assigned to the node as a neutral interval. This neutral interval must be valid, according to the above definition, as there can be no value nodes directly below the given constraint node, that have a value falling into the neutral interval[3]. For example, when the constraint <15 of the tree in Figure 7.1 is reached, the neutral interval ≥15 is assigned to the given constraint node.

In a similar way, a neutral interval $\leq B$ ($\geq B$) can be associated with each max (min) node, which has a best-so-far value B. For example, when the child of the root with the value 20 is reached in our sample

[3]Note that because of the inheritance of neutral intervals this seemingly trivial fact can be utilised for pruning subtrees below the constraint node (see later).

tree, the root's best-so far value becomes 20, and so a neutral interval $\leq$20 can be associated with the root. This can be interpreted as the statement of the following fact: "values $\leq$ 20 are indifferent, i.e. need not be distinguished from each other"[4].

An important property of neutral intervals is that they are inherited by descendant nodes, i.e. if a neutral interval is associated with a node, then it can be associated with any descendant of the node as well. This can be easily proven using the continuity property of intervals, as outlined below.

The only non-trivial case of inheritance is the one when a neutral interval is associated with the parent P of an optimum node N. To prove that the same neutral interval can be associated with node N, let us consider the effect of changing the value of a node directly below N within the given neutral interval (say the value is changed from V_1 to V_2, where both V_1 and V_2 are within the neutral interval). A simple examination of cases shows that if the old value of N or the new value of N is outside the closed interval bounded by V_1 and V_2, then the value of N (and consequently the value of the root) could not have changed. This means that if the value of N changes, it changes within the closed interval bounded by V_1 and V_2, that is within the given neutral interval. Using the premise that this neutral interval is associated with node P, we can conclude that the value of the root is unchanged in this case as well. This finishes the proof that the given neutral interval is inherited by the child node N.

There are basically two types of neutral intervals, ones containing $+\infty$ and the ones containing $-\infty$. Two intervals of the same type can always be replaced by the bigger one. This, together with the inheritance property, means that the worker can keep two actual neutral intervals as part of the search status, when the tree is being traversed (which is analogous to the alpha and beta values of the minimax search).

Neutral intervals can be used to prune the search tree. When a worker reaches a node the constraint of which is subsumed by a currently valid neutral interval, then the tree below the constraint node does not have to be explored, and a single solution with an arbitrary value within the neutral interval can be assumed[5]. Optimum nodes act as special constraint nodes in this respect: a max (min) node with a best-so-far value B is equivalent to a constraint $\geq B$ ($\leq B$).

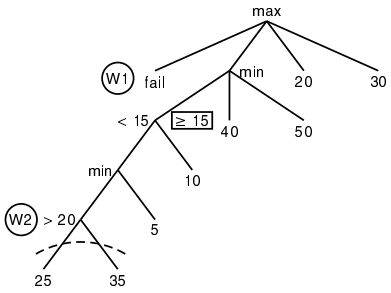

Figure 7.2: First snapshot of exploration of the sample tree

Figure 7.2 shows a snapshot of the exploration of the tree in Figure 7.1 by two workers. Worker w1 has reached the leftmost failure node, while worker w2 descended on the second branch down to the second constraint. Processing of the upper constraint resulted in a neutral interval $\geq$15 being created (shown as a rectangular box in the figure). When the lower constraint of >20 is reached, the worker notices that the constraint is subsumed by the inherited neutral interval and so the subtree below is pruned (as shown by the dotted line).

A second snapshot is shown in Figure 7.3. Worker w1 has now reached the third child of the root, with the value 20. As outlined earlier, this results in a neutral interval $\leq$20 being associated with the root. This neutral interval is now propagated downwards, and its interaction with the constraint <15 results in the whole subtree rooted at that constraint being pruned, i.e a solution with an arbitrary value <15 is assumed (say 0). This example shows why it is necessary to assume an arbitrary solution, instead of discarding the whole subtree. The latter approach would result in an incorrect solution 40 being assigned to the min node, and consequently to the root node as well.

[4]For value nodes directly below the root a stronger statement is valid: "values $\leq$20 can be discarded". For the sake of inheritance, however, the above weaker form is required.

[5]This is the point where we use our simplifying assumption (the existence of a value node below each constraint node).

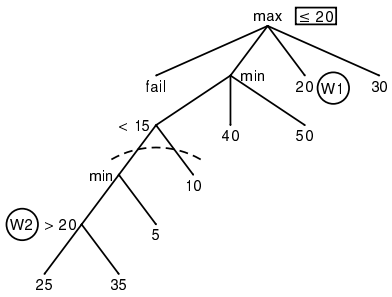

Figure 7.3: Second snapshot of exploration of the sample tree

The propagation of neutral intervals, as exemplified by Figure 7.3, is one of the crucial features of our algorithm. In general, propagation is required when a worker is updating the best-so-far value (and so the neutral interval) of an optimum node, while other workers are exploring branches below this node. The new neutral interval should now be brought to the attention of all workers below the given node. There are two basic approaches for handling this situation:

- The workers below are notified about the new neutral interval, i.e. the information on changes is propagated downwards.
- The downwards propagation is avoided at the expense of each worker scanning the tree upwards every time it wants to make use of the neutral interval (e.g. for pruning).

Reynolds and Kefalas [10] have used the second approach in their proposed extension of the Brave system. A serious drawback of this approach is, however, that it slows down the exploration, even if only a single worker happens to be working on a subtree. Therefore the first approach seems to be preferable, i.e. the worker updating a best-so-far value in an optimum node should notify all the workers below the given node about the new neutral interval.

So far we have assumed that each optimum and constraint node has at least one value node below it. Let us now expand the domain of discussion to include trees where this condition is not enforced. If we extend the range of values that can be associated with nodes to include the infinite values $-\infty$ and $+\infty$, then each failure node can be viewed as a proper value node, with the actual value being $-\infty$ if the optimum node immediately above is a max node, and $+\infty$ if the optimum node immediately above is a min node.

The notion of constraint can have two interpretations in this extended framework. One can consider *strong* constraints, which actually guarantee the presence of a (finite) value node below; and *weak* constraints which may still hold if there is no proper value node in the subtree below. The implicit constraints generated by optimum nodes are obviously of the strong type. On the other hand, not all constraints can be assumed to be strong, as e.g. the constraints used in the branch and bound algorithm are normally of the weak type.

A weak constraint can be utilised (for pruning or for producing a neutral interval) only in one of the two kinds of optimum searches. For example, a weak constraint $< B$ occurring in a minimum search expresses the fact that the contribution of the current subtree to the minimum search will either be a value $< B$, or $+\infty$. This means that such a constraint can not be used to prune the subtree, as it can not guarantee that all values will be part of a single neutral interval. On the other hand a weak constraint of form $> B$ occurring in a minimum search will be equivalent to a strong constraint, and thus can safely be used for pruning, as the "failure" value $+\infty$ is actually part of the constraint interval $> B$.

In our example this means that the first pruning step, shown in Figure 7.2, which is based on the constraint >20 in a minimum search, can be carried out even if the constraint is weak, i.e. if all value nodes below the constraint are replaced by failure nodes. On the other hand, the second pruning step (Figure 7.3) can not be carried out if the the constraint is weak.

7.4 Language Extensions

We propose new higher level predicates to be introduced to encapsulate the algorithm described in the previous section. The optimum search is generalised to allow arbitrary Prolog terms, and an arbitrary ordering relation *LessEq* instead of numbers and numerical comparison. The optimum search returns a pair of terms *Value-Info*, where the *Value* is used for ordering and *Info* can contain some additional information. To simplify the user interface, our experimental implementation assumes all (user-supplied) constraints to be weak.

The proposed new built-in predicates are the following:

`maxof(`*+LessEq*, *?Value-Info*, *+Goal*, *?Max*`)`
`minof(`*+LessEq*, *?Value-Info*, *+Goal*, *?Min*`)`

Max(*Min*) is a *Value-Info* such that *Goal* is provable, and *Value* is the largest (smallest), according to the binary relation *LessEq*, among these *Value-Info* pairs. *LessEq* can be an arbitrary binary predicate, either user-defined or built-in, that defines a complete ordering relation. If *Goal* is not provable, `maxof` and `minof` fails (this failure replaces the infinite values of our abstract algorithm of the previous section). The following example is an illustration for the use of `maxof`:

```
biggest_country(Continent, Country, Area) :-
   maxof(=<, A-C,
        country(Continent, C, A),
        Area-Country).
```

`bestof(`*+Dir*, *+LessEq*, *?Template*, *+Goal*, *?Best*`)`

Dir can be either `max` or `min`. `bestof(max, ...)` is equivalent to `maxof(...)` and `bestof(min, ...)` is equivalent to `minof(...)`. This predicate is just a notational tool for writing minimax-type algorithms.

`constraint(`*?Term1*, *+LessEq*, *?Term2*`)`

Term1 is known to be less or equal to *Term2* according to the binary relation *LessEq*. This means that all solutions of the current branch will satisfy the given condition. One of *Term1* and *Term2* is normally a *Value* of a `maxof`, `minof` or `bestof`, in which case the constraint can be used for pruning. An example:

```
country(europe, Country, Area) :-
   constraint(Area, =<, 600000),
   european_country(Country, Area).
```

`priority(`*+Priority*`)`

Priority should be an integer. This call declares that the current branch of execution is of priority *Priority*. Several calls of the `priority` predicate can be issued on a branch, and the list of these priorities (earlier ones first), ordered lexicographically, will be used when comparing branches. Examples for the use of the `priority` primitive will be given in Section 7.6.

7.5 Implementation

We have designed an experimental implementation of the language primitives described in the previous section, within the current Aurora system itself. This uses a simplified version of the proposed algorithm, as it does not implement the propagation of neutral intervals. The implementation applies the best-first search strategy by default, but depth-first control is also available. This section gives a brief description of the experimental implementation.

Introduction of new control features is normally done via interpretation. We have decided to avoid the extra complexity and overheads of interpretation by introducing a meta-predicate called `task`, to be used to encapsulate the new control primitives within the application program. A call of `task` has the following form:

`task(`*Goal*, *NewContext* - *OldContext*`)`

Here *Goal* is normally a conjunction, which begins with calls of the control predicates `priority` and `constraint`. The invocation of `task` should always be the last subgoal in the surrounding `bestof`. If the *Goal* in `task` contains an embedded call to `bestof`, this should be the last subgoal in the conjunction, to make the minimax algorithm applicable.

The second argument of `task` is required for passing the control information on surrounding tasks and optimum searches. Similarly, the `bestof` (and `maxof`/`minof`) predicates acquire an additional last argument of the same structure. We use the form *NewContext - OldContext* to indicate that the role of this argument is similar to a difference list. The *OldContext* variable links the given call with the surrounding `bestof` or `task` invocation (i.e. it is the same variable as the *NewContext* variable in the extra argument of the surrounding control call). Similarly the *NewContext* variable is normally passed to the *Goal* argument, for use in embedded `task` or `bestof` invocations.

Let us show an example from the previous section in this modified form:

```
country(europe, Country, Area, Ctxt) :-
   task(
        (constraint(Area, =<, 600000),
         european_country(Country, Area)),
        _ - Ctxt).
```

Here we assume, that `european_country` does not contain any further invocations of `task` or `bestof`, hence *NewContext* is a void variable.

The execution of an application in this experimental implementation is carried out as follows. If there are no calls of `task` embedded in a `bestof`, then the optimum search is performed in a fairly straightforward way: a best-so-far value is maintained in the Prolog database which is updated each time a solution is reached.

When an invocation of `task` is reached within the `bestof` predicate, first the constraints are processed: if a constraint indicates that the subtree in question will not modify the best-so-far value (i.e. the constraint is subsumed by the currently applicable neutral interval), then the `task` call fails immediately. Otherwise the goal of the task, paired with information on the constraint, priority and context, is asserted into the Prolog database and the execution fails as well. When all the subtasks have been created and the exploration of the `bestof` subtree finishes, a best-first scheduling algorithm is entered: the subtask with the highest priority is selected and its goal is started. Such a subtask may give rise to further `bestof` and/or `task` calls, which are processed in a similar way.

For the sake of such nested task structure the best-first scheduling is implemented by building a copy of the search tree in the Prolog database, but with the branches ordered according to the user supplied priorities (in descending order). This tree is then used for scheduling (finding the highest priority task), as well as for pruning.

Pruning may be required when a leaf node of the optimum search is reached and the best-so-far value is updated. Following this update the internal tree is scanned and every task, which has become unnecessary according to its constraint, is deleted.

A more detailed description of the implementation can be found in [14].

7.6 Applications

Two larger test programs were developed to help in evaluating the implementation: a program for playing the game of kalah, using alpha-beta pruning, which is based on a version presented by Sterling and Shapiro [13]; and a program for the traveling salesman problem based on the branch-and-bound technique, as described in [1]. This section presents the general program schemes used in these programs, namely the branch-and-bound and alpha-beta pruning schemes. For the sake of readability we omit the additional context arguments in this presentation, but we do include the invocation of the `task` predicate.

7.6.1 The Branch-and-Bound Algorithm

We describe a general program scheme for the branch-and-bound algorithm. We assume that the nodes of the search tree are represented by (arbitrary) Prolog terms. We expect the following predicates to be

supplied by the lower layer of the application:

`child_of(`*Parent*`,` *Child*`)` Node *Child* is a child of node *Parent*.

`leaf_value(`*Leaf*`,` *Value*`)` Node *Leaf* is a leaf node, with *Value* being the value associated with it.

`node_bound(`*Node*`,` *Bound*`)` All leaf nodes below the (non-leaf) node *Node* are known to have a value greater or equal to *Bound*.

```
% Leaf is the leaf below node with the minimal Value
branch_and_bound(Node, Leaf, Value):-
   minof(=<, V-L,  leaf_below(Node, L, V),  Value-Leaf).

% Node has a Leaf descendant with value Value
leaf_below(Node, Node, Value):-
   leaf_value(Node, Value).
leaf_below(Node, Leaf, Value):-
   child_of(Node, Child),
        node_bound(Child, Bound),
        Priority is -Bound,
                                      task((
        constraint(Bound, =<, Value),
        priority(Priority),
   leaf_below(Child, Leaf, Value)           ))
   .
```

Figure 7.4: The general scheme for the branch-and-bound algorithm

Figure 7.4 shows the top layer of the branch-and-bound scheme based on the above predicates. The program uses a single `minof` call invoking the predicate `leaf_below(Node, Leaf, Value)`. The latter predicate simply enumerates all the `Leaf` nodes and corresponding `Value`s below `Node`. The logic of this predicate is very simple: either we are at a leaf node (first clause), in which case we retrieve its value, or we pick up any child of the node and recursively enumerate all the descendants of that child (second clause). This logic is complemented with the calls providing the appropriate control (shown with a deeper indentation): calculating a lower bound for the relevant subtree (`node_bound`), calculating the `Priority` as the negated value of `Bound` (so that the subtrees where the bound is lower have higher priority), notifying the system about the bound (`constraint`) and the priority for the best-first search (`priority`). The last three calls in the clause are encapsulated within the auxiliary predicate `task` (shown with the deepest indentation).

This general scheme of Figure 7.4 can be concretised to support a specific application by designing an appropriate node data structure, and providing the definition of the lower level predicates (`child_of` etc.). This has been done for the traveling salesman problem, the preliminary performance results for which are presented in Section 7.7.

7.6.2 The Alpha-Beta Pruning Algorithm

We now proceed to describe a similar scheme for the minimax algorithm with alpha-beta pruning (Figure 7.5). Again we allow the nodes of the game tree to be represented by arbitrary Prolog terms. The topology of the tree and the values associated with nodes are expected to be supplied through predicates of the same form as for the branch-and-bound algorithm (`child_of(`*Parent*`,` *Child*) and `leaf_value(`*Leaf*`,` *Value*`)`). We require two additional auxiliary predicates:

`node_priority(`*Node*`,` *Prio*`)` *Prio* is the priority of node *Node*.

`absolute_min_max(`*Min*`,` *Max*`)` *Min* and *Max* are the absolute minimum and maximum values for the whole of the game tree[6].

```
% Node of type Type (min or max) has the value Value,
% produced by Child.
alpha_beta(Node, Type, Child, Value):-
   bestof(Type, =<, V-C,
          child_value(Node, Type, V, C),  Value-Child).

% Node of type Type has a Child with Value.
child_value(Node, Type, Value, Child):-
   opposite(Type, OppType),
       absolute_min_max(Min, Max),
                                                  task((
       constraint(Min, =<, Value),
       constraint(Value, =<, Max),
   child_of(Node, Child),
   node_value(Child, OppType, Value)                    ))
   .

% Node of type Type has Value.
node_value(Node, _, Value):-
   leaf_value(Node, Value).
node_value(Node, Type, Value):-
       node_priority(Node, Priority),
                                                  task((
       priority(Priority),
   bestof(Type, =<, V-null,
          child_value(Node, Type, V, _),  Value-null)    ))
   .

opposite(max,min).
opposite(min,max).
```

Figure 7.5: The minimax algorithm with alpha-beta pruning

This scheme can be invoked by the `alpha_beta(Node, max, Child, Value)` call. Here `Node` represents a node of the game tree, and `max` indicates that this is a maximum node. The call will return the `Child` with the maximal `Value`, from among all children of `Node`.

The `alpha_beta` predicate is defined in terms of a `bestof` search over all `Child-Value` pairs enumerated by the `child_value` predicate. This predicate, in its turn, issues appropriate constraint directives, enumerates the children (`child_of`), and invokes `node_value` for every child. The `node_value` predicate has two clauses, the first is applicable in the case of leaf nodes, while the second invokes the opposite `bestof` over `child_value` recursively, after having informed the system about the priority applicable to the given subtree.

Note that the algorithm presented in Figure 7.5 calculates the optimum with respect to the complete game tree. It is fairly easy, however, to incorporate an appropriate depth limit, as usually done in game playing algorithms, by a simple modification of this scheme.

7.7 Performance Results

Table 7.1 gives some early performance figures for the applications discussed in Section 7.6, using the experimental implementation described in Section 7.5.

The tests have been run on a Sequent^{TM} Symmetry S27 multiprocessor with 12 processors, and the Manchester scheduler [5] has been used. Time (in seconds) is given for the one-worker case, and speedups are shown for 2-10 workers.

[6] Note that the scheme is still usable if no such absolute bounds are available—one just has to delete those parts of the program, which deal with the constraints based on the absolute bounds.

Version	Workers					
	1	2	4	6	8	10
	(Time)	(Speedup)				
Traveling salesman						
9 nodes	32.2	1.68	2.66	3.24	3.71	4.07
11 nodes	152.86	1.64	2.67	3.48	3.93	4.37
The game of kalah						
board 1	13.71	1.39	1.83	2.35	2.96	3.32
board 2	57.33	1.71	2.88	3.75	4.72	5.65
board 3	31.20	1.70	3.08	4.04	5.18	5.67
board 4	65.66	1.55	2.47	3.66	4.73	5.00

Table 7.1: Run times and speedups for various optimisation problems

The two traveling salesman sample runs involve complete graphs with 9 and 11 nodes, and 36 and 55 edges, respectively. A variant of the game of kalah is used as the second test program. Four different board states are tested with a limited depth of search (4 steps). Because of the search tree being so shallow, the depth-first strategy is used, rather than the best-first one.

Considering the prototyping nature of our experimental implementation we view the results as quite promising. We plan to carry out a detailed performance evaluation in the near future to identify the overheads involved in various parts of the algorithm.

7.8 Related Work

An important issue is the relation of our work to the mainstream of research in constraint logic programming (CLP). In current CLP frameworks (as e.g. in the one described by van Hentenryck in [7]) the constraints arising in optimum search algorithms are handled by special built-in predicates. The reason behind this is that the generation of constraints is implicit in an optimum search, as the applicable constraint depends on the best-so-far value. We believe that by replacing such special predicates with the `bestof` construct, our extended algorithm can be smoothly integrated into a general CLP system.

Another aspect of comparison may be the type of parallelism. Current CLP systems address the issues of and-parallel execution of conjunctive goals as e.g. in the CLP framework described by Saraswat [12]. Our approach complements this by discussing issues of exploiting or-parallelism. Combination of the two types of parallelism can lead to much improved performance as shown by existing and-or-parallel systems, such as Andorra [11].

The problems of or-parallel execution of optimum search problems have been addressed by Reynolds and Kefalas [10] in the framework of their meta-Brave system. They introduce a special database for storing partial results or *lemmas*, with a restricted set of update operators. They describe programs implementing the minimax and branch-and-bound algorithms within this framework. They do not, however, address the problem of providing a uniform approach for both optimisation algorithms. Another serious drawback of their scheme is that pruning requires active participation of the processing agent to be pruned: e.g. in the minimax algorithm each processing agent has to check all its ancestor nodes, whether they make further processing of the given branch unnecessary.

7.9 Conclusions

The design and the implementation of the `bestof` predicate has several implications. First, we have developed a new higher order extension to Prolog, with an underlying algorithm general enough to encapsulate two important search control techniques: the branch-and-bound and alpha-beta pruning algorithms. The `bestof` predicate makes it possible to describe programs requiring such control techniques, in terms of special control primitives such as constraints and priority annotations. On the other hand we gained important experience by implementing the new predicates on the top of Aurora system. We believe that this experience can be

utilised later, in a more efficient, lower level implementation as well.

We view the development of the `bestof` predicate as a first step towards a more general goal: identifying those application areas and special algorithms where the simple control of Prolog is hindering efficient parallel execution, and designing appropriate higher order predicates that encapsulate these algorithms. We believe that the gains of this work will be twofold: reducing the need for non-declarative language components as well as developing efficient parallel implementations of such higher order primitives.

Acknowledgements

The author would like to thank his colleagues in the Gigalips project at Argonne National Laboratory, the University of Bristol, the Swedish Institute of Computer Science and IQSOFT. Special thanks go to David H. D. Warren for continuous encouragement and help in this work. Thanks are also due to the anonymous referees, for valuable comments and suggestions for improvement.

This work was supported by the ESPRIT project 2025 "EDS", and the Hungarian-U.S. Science and Technology Joint Fund in cooperation with the Hungarian National Committee for Technical Development and the U.S. Department of Energy under project J.F. No. 031/90.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] Khayri A. M. Ali and Roland Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceedings of the North American Conference on Logic Programming.* The MIT Press, October 1990.

[3] 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.

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

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

[6] J. Chassin de Kergommeaux and P. Robert. An abstract machine to implement efficiently OR-AND parallel Prolog. *Journal of Logic Programming*, 7, 1990.

[7] Pascal van Hentenryck. *Constraint Satisfation in Logic programming.* The MIT Press, 1989.

[8] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[9] B. Ramkumar and L.V. Kalé. Compiled execution of the reduce-OR process model on multiprocessors. In *Proceedings of the North American Conference on Logic Programming*, pages 331–331. The MIT Press, October 1989.

[10] T. J. Reynold and P. Kefalas. OR-parallel Prolog and search problems in AI applications. In *Logic Programming: Proceedings of the Seventh International Conference*, pages 340–354. The MIT Press, 1990.

[11] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the Eighth International Conference.* The MIT Press, 1991.

[12] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989.

[13] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[14] Péter Szeredi. Design and implementation of Prolog language extensions for or-parallel systems. Technical Report, IQSOFT and University of Bristol, December 1990.

[15] Péter Szeredi. Using dynamic predicates in an or-parallel Prolog system. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Logic Programming Symposium*, pages 355–371. The MIT Press, October 1991.

[16] Péter Szeredi. Exploiting or-parallelism in optimisation problems. In Krzysztof R. Apt, editor, *Logic Programming: Proceedings of the 1992 Joint International Conference and Symposium*, pages 703–716. The MIT Press, November 1992.

[17] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

Part III

Applications

Chapter 8

Applications of the Aurora Parallel Prolog System to Computational Molecular Biology[1]

Ewing Lusk

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

Shyam Mudambi

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

Ross Overbeek

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

Péter Szeredi

IQSOFT Ltd.
H-1142 Budapest Teleki B. 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.

8.1 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.

[1] This paper has appeared in the proceedings of ILPS'93 [9]

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.

8.2 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.

8.3 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.

8.3.1 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 [10], 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 [10]. 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.

8.3.2 Visualization of Parallel Logic

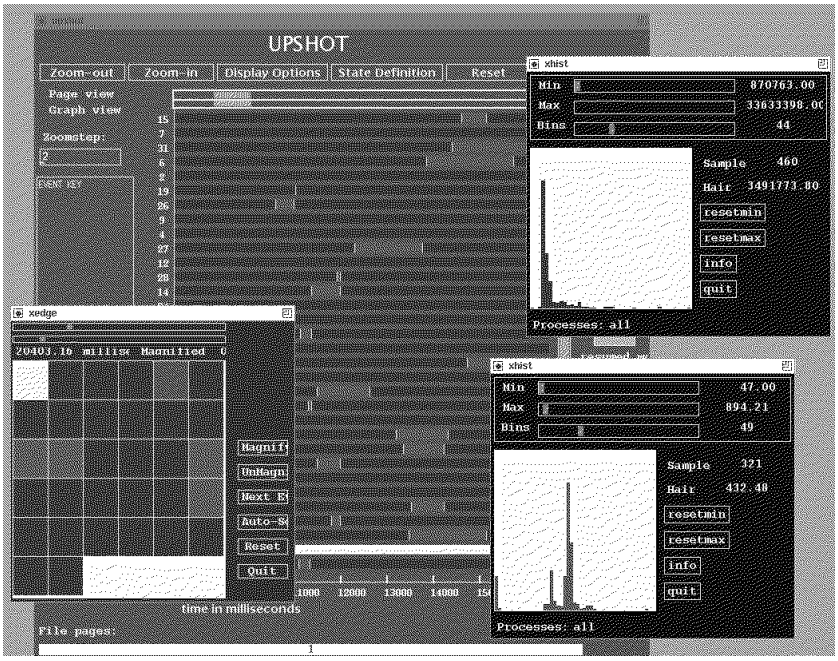

Figure 8.1: Investigating grain size with `upshot`

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 8.1. 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 8.3.

8.4 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].

8.4.1 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 8.2.

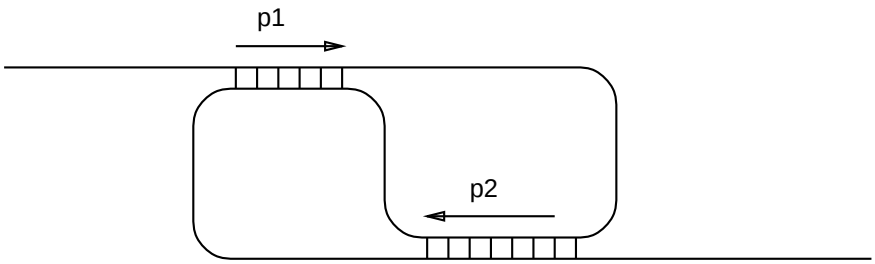

Figure 8.2: A pseudo-knot

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 [11]) 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.

8.4.2 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".

8.5 Evaluation of Experiments

8.5.1 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 single 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 8.2) 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.

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]

Table 8.1: Pseudo-knot Queries

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)

Table 8.2: Results of pseudo-knot query.

The specific pseudo-knot queries used in our tests are shown in Table 8.1. The times in seconds for these queries, run with varying numbers of processes on the TC-2000, are shown in Table 8.2. 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 8.3. The various shades of gray (colors on a screen) indicate states of the Aurora parallel abstract machine. This particular picture indicates that 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 8.3 shows the speed improvements obtained by pre-sorting sequence-fragments for selected queries.

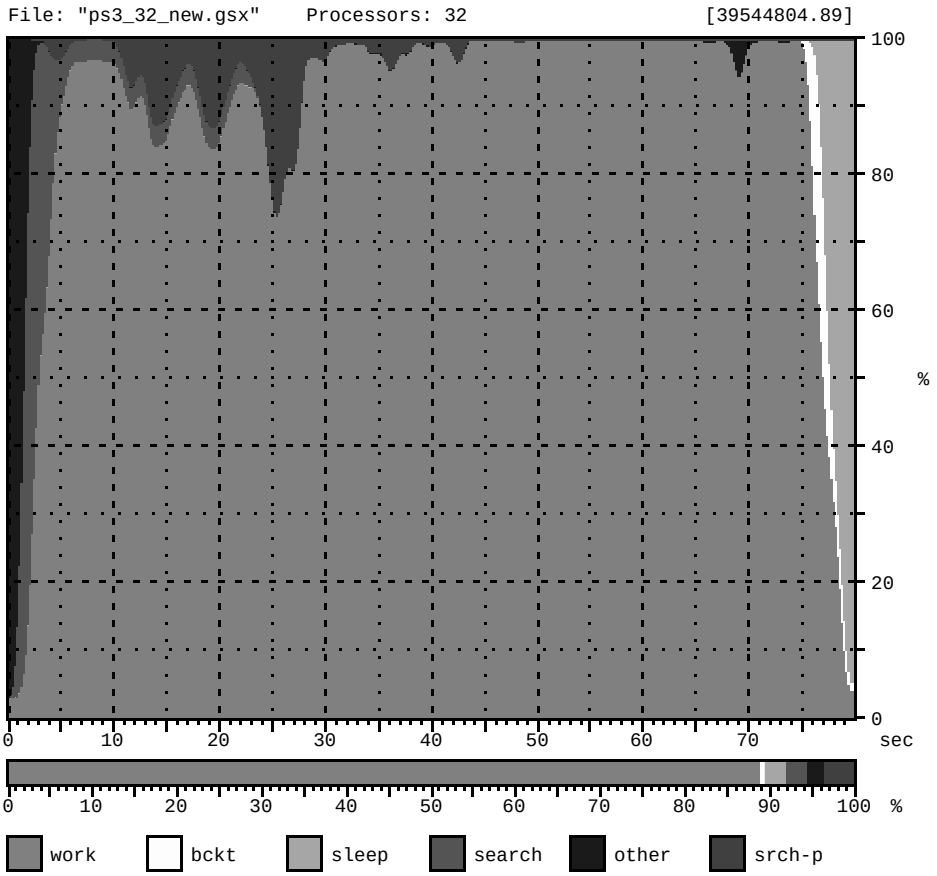

Figure 8.3: Efficiency of pseudo-knot query on TC-2000

They are small but definitely significant.

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%

Table 8.3: Percentage improvements due to sorting

8.5.2 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 probability 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 8.4.2. 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 8.4.

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 8.4: 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 8.4 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 8.4, 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 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 8.4. The speedups are almost linear, being less than 1% below the ideal speedup. Although this change is enough on its own

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					

Table 8.4: Exploring the Search Space in the Protein Motifs Query

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[2], 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 8.5. 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 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.

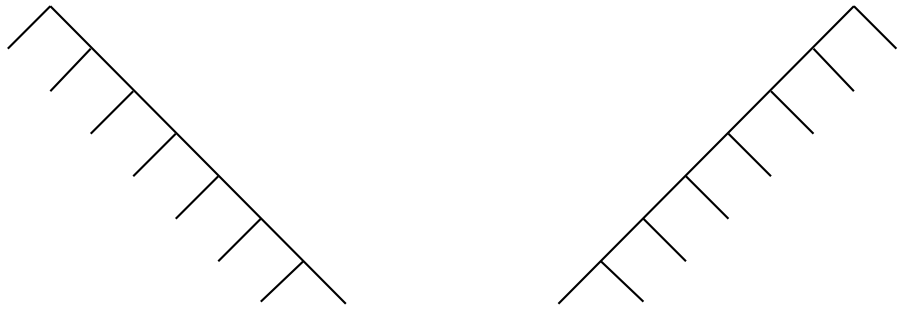

Figure 8.5: Alternative parallel search trees

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[3].

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,

[2] We just show the search aspects of the program, omitting those parts which deal with returning the solution.
[3] We actually used the Bristol scheduler with bottommost scheduling.

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 8.5. 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 8.4 shows the effect of bottom-to-top exploration 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 8.5. The very first row, showing the times for the failure driven loop, is identical to the last row of Table 8.4 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%.

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)

Table 8.5: Results of the Protein Motifs Query

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 8.5). Our visualization and tuning tools (Figure 8.1) 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 8.5. 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 8.5).

8.6 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 8.5.1) 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.

Acknowledgements

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] Ewing Lusk, Shyam Mudambi, Ross Overbeek, and Péter Szeredi. Applications of the Aurora parallel Prolog system to computational molecular biology. In Dale Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 353–369. The MIT Press, November 1993.

[10] 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.

[11] 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.

Chapter 9

Handling large knowledge bases in parallel Prolog[1]

Péter Szeredi and Zsuzsa Farkas

IQSOFT Intelligent Software Co. Ltd.
H-1142 Budapest, Teleki Blanka utca 15-17, Hungary
{szeredi,farkas}@iqsoft.hu

Abstract

The paper describes our work on using parallel Prolog systems for implementing relatively large hierarchical knowledge bases, carried out within the CUBIQ Copernicus project.

The CUBIQ project is an attempt to integrate three strands of computing: expert systems, parallel computing and graphical interaction and visualisation. For core expert system development the CUBIQ basic tool-set has been developed; parallel computing is supported by two or-parallel Prolog systems: Aurora and Muse; and the graphical tools have been developed using the ICD-Edit 3-D diagrammatical interaction tool and the Tcl/Tk library.

The problem of representing and using a large hierarchical knowledge base arose within the EMRM (Electronic Medical Record Management) system, one of the prototype applications developed in the CUBIQ project. This application uses the SNOMED hierarchical medical thesaurus.

The paper describes our experiments with several alternative representation techniques used for implementing the SNOMED hierarchy of the EMRM system. We also describe how the results of these experiments influenced the development of the CUBIQ tool-set itself. We present parallel performance results for typical searches within the SNOMED tree hierarchy for both Aurora and Muse.

Keywords: Prolog, expert systems, parallel execution, Aurora, Muse, SICStus, Prolog Objects, SNOMED

9.1 Introduction

The CUBIQ Copernicus project aims at the integration of three strands of computing: expert systems, parallel computing and graphical interaction and visualisation [4]. The main objective of the project is to produce a Prolog-based tool-set (called the CUBIQ tool-set), with features to aid expert system development

[1] This report has been presented at the Workshop on High Performance Logic Programming Systems [9]

and graphical interaction, which also supports parallel execution. The implementation of CUBIQ uses SICStus Prolog and its or-parallel extensions Aurora and Muse [5, 1]. The graphical components of the system are based on the ICD-Edit 3-D diagrammatical interaction tool [3] and the Tcl/Tk library. More details on the graphical aspects of CUBIQ can be found in [10].

The partners in the CUBIQ project are IQSOFT (Budapest, Hungary), University of Bristol (UK) and City University (London, UK).

The CUBIQ project includes the development of two prototype applications of the tool-set: the CONSULT credit rating expert system and the EMRM electronic medical record management system. This paper deals with the medical thesaurus component of the EMRM application.

The medical thesaurus is based on SNOMED (Systematized Nomenclature of Medical Knowledge), [7]. The SNOMED thesaurus contains approximately 40,000 medical phrases arranged into a tree hierarchy. This tree structure plays a vital role in two major functions of the EMRM system:

browsing: selecting an appropriate SNOMED phrase using a substring of its (English) name, and

inheritance in diagnosis: finding a diagnosis rule applicable to a medical object by inheritance in the SNOMED hierarchy.

The CUBIQ tool-set introduces the notion of frames and frame inheritance to support hierarchical knowledge representation. The main implementation of the SNOMED hierarchy presented in this paper is built on top of CUBIQ frames.

The paper describes our experiences in the design and implementation of the SNOMED component of EMRM with special attention paid to issues of parallel performance. Section 9.2 gives basic background information on CUBIQ, EMRM and the parallel Prolog systems used in the project. Section 9.3 outlines our initial experiments on implementing SNOMED in plain Prolog and an existing library extension (Prolog Objects). Section 9.4 describes the evolution of the frame representation in CUBIQ, as influenced by the EMRM application and its performance analysis. Section 9.5 describes our performance measurements for both Aurora and Muse. Finally Section 9.6 summarises the conclusions of the paper and outlines future work.

9.2 Background

9.2.1 The CUBIQ tool-set

The CUBIQ basic tool-set is built on top of (SICStus) Prolog. Several language extensions (based on `term_expansion/2`) are provided to support the implementation of knowledge bases. The main extensions are frames, functional rule notation, uncertainty handling [6], and memoisation. All these extensions are properly integrated with each other.

This section focuses on outlining the main features of CUBIQ frames, for a more detailed description of the system see [4, 10].

The bottom layer of CUBIQ is a frame-extension of Prolog. The frames are objects of the world to be modeled, arranged in a parentship hierarchy. Relations can be defined on one or more frames, and then these definitions are inherited from the more general to the more specific frames.

CUBIQ uses the concept of the *relation* as a primary notion. That is, relations may be defined with frames as arguments. Formally, frames are introduced with a declaration of the following form:

```
:- frame(Frame, Parents, Attribute).
```

where the `Frame` atom is a frame identifier, `Parents` specifies the parent(s) of `Frame`, being either a frame identifier or a list of frame identifiers, and the optional `Attribute` is an arbitrary term, attached to `Frame` as a (non-inherited) attribute.

Relations attached to frames are defined by so-called *frame clauses* containing frame references of the form `frame::X`. An argument of this form appearing in a clause head means that the clause applies to all descendants `X` of frame `frame`. Accordingly, when an argument of the form `::frame` appears in a goal, the

corresponding predicate definition is sought among all the ancestors of `frame`. We also allow `ancestor::X` as a goal argument, in which case `X` is first instantiated to a descendant of `ancestor` (and all descendants are enumerated on backtracking), and for each such instantiation of `X`, the inheritance search is used to determine the applicable definition.

CUBIQ frame handling is illustrated with a simple example in Figure 9.1.

```
:- frame(animal,[]).
:- frame(carnivore,animal).
:- frame(herbivore,animal).
:- frame(fox,carnivore).
:- frame(rabbit,herbivore).

size(rabbit::_, 8).
size(fox::_, 30).

eats(carnivore::C, herbivore::H) :-
     size(::C,S1),              % A carnivore eats a
     size(::H,S2),              % herbivore if it is
     S1 >= 2 * S2.              % at least twice as big.

% A sample invocation:
?- eats(animal::X, animal::Y).

       X = fox
       Y = rabbit
```

Figure 9.1: A simple example illustrating CUBIQ frames

By default a *multiple, overriding* inheritance mechanism is used for frame relations, but the user can also supply an alternative inheritance mechanism.

Note that the CUBIQ tool-set supports dynamic frame handling as well, but the application problem discussed in this paper does not require this feature. This is important because, in order to preserve sequential Prolog semantics, most or-parallel systems execute dynamic predicates in a synchronised way, thus accruing large overheads and losing advantages of parallel execution.

9.2.2 EMRM: a medical application with a large medical thesaurus

One of the prototype applications of the CUBIQ project chosen for testing the practical usability of its components is the EMRM (Electronic Medical Record Management) system. The goal of this application is to give knowledge-based assistance for the physician in the patient-related administration process. The knowledge representation of EMRM is built on top of the SNOMED hierarchical medical thesaurus. SNOMED is a structured nomenclature and classification of the terminology used in human and veterinary medicine, covering all important aspects of medicine. It is supported by the American Medical Society, and it is one of the main candidates for emerging medical terminology standards.

SNOMED constitutes a large though rather flat hierarchy: it contains about 40,000 nodes (but more than 90% of these are leaf nodes, and the maximal depth is 5). Each SNOMED node has a unique (hexadecimal) code and contains a reference to its parent SNOMED code as well as a number of attributes. The most important attribute is the textual description of the notion represented. Nodes can have further attributes, such as alternative descriptions (aliases), references to other SNOMED nodes, etc. In a somewhat simplified way a SNOMED node can thus be characterised by the following data:

```
<SNOMED-code> <parent SNOMED-code> <attributes>
```

The SNOMED thesaurus is divided into parts (called modules) according to the type of phrase in question. The biggest SNOMED module is that of the diseases, other modules include living organisms, chemicals, morphology, topography etc.

In the internal data structures of EMRM each medical phrase is represented by the SNOMED-code, possibly refined using some free text qualifiers. When information from such data structures is displayed, the textual description is substituted for the code. Input of medical data is supported by the *SNOMED browser* which is a tool for navigating around the SNOMED thesaurus, combining browsing in the hierarchy with text search. Using the browser the user can arrive at the node representing the medical phrase to be entered without knowing the exact text associated with the node.

Further to the SNOMED browser, the other main component of EMRM where the SNOMED hierarchy plays a crucial role is the *diagnosis support* function. Here the information inheritance principle is applied to support refinement of diagnoses. For example, in EMRM there is a rule expressing that a generic disease group is indicated as a possible diagnosis by the family history, if a disease of that type (i.e. a SNOMED-descendant of the generic disease) has already appeared in the close family. Figure 9.2 shows how such a rule can be expressed using CUBIQ frames. Implementation of such rules again requires efficient search in the SNOMED tree hierarchy.

```
possible_diagnosis(disease::DiseaseGroup) :-
     relative_with_recurrent_disease(disease::DiseaseGroup, Relative),
     close_relative(Relative).

relative_with_recurrent_disease(disease::DiseaseGroup, R) :-
     recurrent_disease_in_family_history(DiseaseGroup::Disease, R).

recurrent_disease_in_family_history(disease::Disease, R) :-
     history_of_relative(R, Disease, recurrent, _).
```

Figure 9.2: Diagnosis search using CUBIQ frames

For our parallel performance analysis we have selected four search problems, two from the browser and two from the diagnosis refinement component. These are discussed in more detail in Section 9.5.

9.2.3 Or-parallel Prolog systems used in CUBIQ

The main parallel Prolog platform used in the CUBIQ project is Aurora. Aurora [5] is an or-parallel implementation of full Prolog based on the SICStus 0.6 engine. Several schedulers were developed for Aurora, in the present project the Bristol scheduler was used [2].

In the final phase of the project the Muse [1] or-parallel implementation based on SICStus 3 was released. We included the Muse system in our evaluation for comparison with Aurora, and also for experiments with some newer SICStus features (such as Prolog Objects) which were not supported by the Aurora engine.

Aurora uses the binding array approach of the SRI model [11] to represent multiple bindings in or-parallel search, while Muse uses the copying approach for solving this problem. Although both systems are based on SICStus, they use different versions: the Aurora engine is older and slower than that of Muse. On the other hand Aurora supports a number of extensions, such as the commit pruning operator, non-synchronising input-output and database operations.

In order to provide a fair comparison our experiments use only the common part of the two systems. In the process of program development we tried to eliminate the use of features for which the speed of the two systems significantly differ. For example, the foreign interface of SICStus 0.6 is much slower than that of SICStus 3, while atom composition and meta-calls turned out to be slower in SICStus 3 than in SICStus 0.6.

9.3 Representing the SNOMED hierarchy in Prolog

Selecting the right representation for the SNOMED hierarchy was a crucial issue in the design of the EMRM system. At first, considering the size of this thesaurus, it seemed natural to use an external data base for storing SNOMED. There were, however, strong arguments for representing SNOMED within Prolog: its hierarchical structure called for a hierarchical knowledge representation technique which can be easily built on top of Prolog. The Prolog representation also allowed us to explore the issues of or-parallel execution of the SNOMED search.

In this section we outline our experiments with a representation in plain Prolog and in Prolog Objects.

Our first experiment was to transform the SNOMED thesaurus into a set of plain PROLOG clauses, to test the feasibility of such a representation and to test whether or not SICStus Prolog was suitable for handling this large number of atoms and clauses.

As a first attempt, the following trivial Prolog representation was chosen for SNOMED nodes:

```
snomed(<SNOMED-code>, <parent SNOMED-code>, <attributes>).
```

Here the `attributes` part is a structure of the form `attr(Name,Aliases,OtherAttrs)`. An example of the Prolog representation of a SNOMED node is shown in Figure 9.3.

```
snomed('D0-01150',
       'D0-01100/00',
       attr('Furuncle of skin and subcutaneous tissue, NOS',
         ['Boil of skin and subcutaneous tissue, NOS'], [])).
```

Figure 9.3: An example for simple representation of SNOMED nodes

We were able to compile and load the above Prolog representation of the whole SNOMED thesaurus into SICStus 3, but not into Aurora, as the load time of large predicates (i.e. ones consisting of a large number of clauses) in the SICStus 0.6 based engine of Aurora is prohibitively large.

The speed of search with the above primitive representation was not satisfactory in SICStus 3 either. This was partly due to the single-predicate representation, but also due to limited indexing: as SICStus has indexing on the first argument only, the operation of finding children of a parent SNOMED node is very expensive in the above representation.

It became obvious that the representation should be improved, by splitting up the large predicate into a set of smaller ones, and also catering for fast access to children. However, we decided not to continue the development of the Prolog representation for the specific case of SNOMED thesaurus. Instead, in conformance with the goals of the CUBIQ project, we proceeded to transform the generic frame representation of CUBIQ in such a way that (parallel) handling of large hierarchical structures, such the SNOMED thesaurus, became feasible.

Before embarking on this task, we have made an additional experiment, regarding the feasibility of using an existing object-oriented extension of SICStus Prolog, Prolog Objects, for representing SNOMED.

A straightforward representation, mapping each SNOMED node to an object is shown in Figure 9.4.

```
<SNOMED-code> ::
       {
        super(<parent SNOMED-code>) &
        attributes(<attributes>)
       }.
```

Figure 9.4: Representation of SNOMED nodes in Prolog Objects

It turned out that with this representation the memory requirements are prohibitively large when loading the disease part of the SNOMED thesaurus. This is because each object becomes a fully-fledged SICStus module, with significant memory overheads.

The newest release of Prolog Objects introduces the notion of object instance. As most of the nodes of the SNOMED tree are leaves, this feature can be used to reduce storage requirements by using instances to represent SNOMED leaves. While in the earlier representation the built-in `descendant` method could be directly used to enumerate all the descendants, in the new version we had to define our own method for this purpose, using the `descendant` method inside the tree, and the `has_instance` built-in for leaves.

The new representation allowed us to load the whole of the SNOMED disease module. The run-time of a search involving scanning all the diseases was still prohibitively large. In our understanding, this is due to

the complexity of object representation and rather crude implementation of certain primitives. For example the `has_instance` primitive actually enumerates all the instances of the "world" and then filters out the instances of the given object.

We have experimented with the parallel behaviour of SNOMED searches in Prolog Objects using the Muse or-parallel implementation. We have experienced a 10% slowdown for multiple processors (irrespective of their number). This is clearly due to the fact that the SICStus Prolog Objects enumeration predicates were not created with parallel execution in mind. The enumeration process of (non-instance) children of an object is based on a dynamic predicate, while the enumeration of instances of an object reduces to the `current_module` built-in of SICStus, which, in turn, also relies on a dynamic predicate. The compulsory synchronisation of dynamic predicates in Muse practically prohibits parallel execution of the search in the SICStus Prolog Objects inheritance tree structure.

9.4 The evolution of the frame representation in CUBIQ

This section describes the evolution of the CUBIQ frame representation format directed towards supporting large frame hierarchies and their parallel search. In this process the following efficiency factors were considered:

- execution speed (sequential and parallel),
- program size,
- development speed (time of consult, compile and load).

The initial representation for frames was fairly straightforward, a frame declaration of the form:

```
:- frame(Frame, Parents, Attribute).
```

was transformed into a clause:

```
static_frame(Frame, Parents, Attribute).
```

This directly corresponds to the initial SNOMED representation discussed in Section 9.3: a single predicate stores the whole frame inheritance structure. This representation produced acceptable execution speeds for hierarchies up to a few hundred frames, but for hierarchies with tens of thousands of frames, the drawbacks became apparent. Further to execution speed problems, the use of a single large predicate caused huge slowdown in compilation and load time for Aurora.

To remedy these problems, an alternative representation was developed, avoiding large predicates and providing faster access to the children of a frame. In this representation a frame was translated to a clause of the form:

```
Frame(Parents, Children, Attribute).
```

Note that the name of the frame was used as the predicate name, so that each frame declaration was transformed into a separate predicate[2]. For example, the frames of the example in Figure 9.1 were transformed into a set of clauses shown in Figure 9.5.

With this representation we got a marked improvement in execution speed, however, for large frame structures the size of the code became very big. Obviously, the detrimental effects were due to the large number of additional predicates generated for frames.

We have thus explored two extreme representation schemes for frames: all frames stored in a single predicate and a separate predicate for each frame. Both schemes proved to be unacceptable from some aspects, hence a compromise between the two approaches had to be sought.

[2] A minor restriction associated with this solution is that frame names are not allowed to appear as predicate names in the user program (at least with arity 3).

```
% Frame(Parents, Children, Attrs).
animal([], [carnivore,herbivore], []).
carnivore([animal], [fox], []).
herbivore([animal], [rabbit], []).
fox(carnivore, [], []).
rabbit(herbivore, [], []).
```

Figure 9.5: Optimised representation of CUBIQ frames of Fig. 9.1

An obvious compromise is to split the set of frames into groups, and have each group represented by a single predicate, where each clause of the predicate represents a frame within the group. In general form, let us assume there exists a $f(Frame)$ hash-function, which maps a frame name to the corresponding group name. The general frame representation scheme based on the grouping implied by the f function is thus the following:

f`(Frame)(Frame, Parents, Children, Attribute).`

The hash function f has to be fairly cheap to evaluate and has to map atoms to atoms. As a first candidate, we selected a very simple such function, which maps an atom into (an atom composed of) its first three characters. Note that in the case of the SNOMED disease database, due to the format of SNOMED codes, two of the three initial characters of frame names are fixed. This solution thus resulted in partitioning the SNOMED disease representation into just 16 predicates. Although we got some slowdown in execution time with regards to the previous approach, the space overhead with respect to the single predicate representation turned out to be negligible, and the development efficiency became acceptable.

Although the simple "name slicing" approach proved to be very good for the case of the SNOMED frame hierarchy, it is very much dependent on the actual naming of the frames. Therefore we subsequently tried a "real" hash function using the `term_hash` predicate of the SICStus `terms` library with moduluses 17 and 257[3]. We got very similar development efficiency and size, and somewhat faster execution times. Detailed time and space figures and the analysis of the results are given in the next section.

A further issue arising in parallel execution of frame hierarchy searches is that of the handling of dynamic frames. It was clear from the beginning that it is feasible to separate the representation of static and dynamic frames, so that the former ones can be compiled and executed in parallel. The tool-set functions accessing the frame structure thus have to look at both static and dynamic parts. Note, however, that in parallel Prolog systems that aim to preserve sequential semantics even an attempt to access an empty dynamic predicate definition will cause synchronisation and thus kill the parallelism. We have therefore introduced a load-time switch into the CUBIQ system, by which the user can assert that no dynamic frames will be used. With this switch set, the system disallows dynamic frames and searches using the static frame structure only.

9.5 Performance analysis of SNOMED searches

In this section we present and analyse sequential and parallel performance of SNOMED searches for both Aurora and Muse. All measurements were carried out on a Sequent Symmetry with six 486/50MHz processors, running Dynix ptx 2.1.1.

We use the following four benchmark problems, listed in increasing size:

- `diagn1`: Check the presence of the descendants of three typical disease groups in a given family history.
- `diagn2`: Check the presence of all diseases in a given family history.
- `browse1`: Browse the whole tree hierarchy of the disease module looking for a node whose name or aliases contain a given substring.

[3] We have moved the hashing algorithm (coded in C) of this SICStus 3 library predicate to Aurora with practically no change, in order to be able to use it in Aurora as well.

- `browse2`: Browse the whole disease hierarchy with a complex search term requiring the checking of three substring matches, their results being combined using `and` and `or` boolean operators.

All these searches are concerned with the disease SNOMED module, containing approximately 18,500 SNOMED terms, represented by the same number of clauses. About 45,000 different Prolog atoms appear in the representation of the disease module. The or-parallelism comes from the possibility of exploring alternative branches of the SNOMED tree in parallel.

The diagnosis and browse benchmark groups are of a slightly different nature. The former are dominated by the tree search proper, with very little work to be done for most of the tree nodes. Furthermore, the `diagn1` search problem is very small, as it is concerned with only a part of the disease tree (`diagn2` scans the whole disease tree).

The browse searches also scan the whole disease tree and do some non-trivial string (atom) processing tests for each node. The basic test for checking whether an atom contains another one (ignoring case) is implemented in C, as a new built-in, in both Aurora and Muse[4]. In `browse1` this basic test is run for the node name and its aliases, while in `browse2` a small boolean expression evaluator is invoked with the "atom contains" test at the bottom. The browse searches are thus of coarser granularity than the diagnosis ones.

9.5.1 Sequential performance

In this section we discuss basic sequential performance of SNOMED disease hierarchy searches, using various frame representations in both Aurora and Muse.

Note that when comparing sequential behaviour of Aurora and Muse, most of the time we are actually comparing their engines (SICStus 0.6 and SICStus 3)[5]. As we have no access to SICStus 0.6 at the moment, we decided to use the single processor versions of the parallel systems for sequential comparison.

Frame representation	Avrg pred size	Muse				Aurora
		Comp. time (sec)	Load time (sec)	Size (Mb)	`diagn1` exec. time (sec)	
1. *single* predicate	18500	331	593	5.2	502	868
2. *many* predicates	1	401	37	11.4	0.93	0.74
*many** (simpl. meta-calls)					0.44	
3. grouping by *prefix* of *3* chars	1150	393	40	5.2	1.75	1.16
*prefix 3** (no meta-calls)					1.07	0.93
4. grouping by *hash* mod *17*	1100	391	46	5.4	0.51	0.91
5. grouping by *hash* mod *257*	72	401	21	5.4	0.51	0.94

Table 9.1: Comparison of the frame representation schemes

Table 9.1 gives an overview of sequential performance data, with its rows corresponding to various frame representation schemes. Words printed in italics in the row headings are used to identify the scheme in the sequel. The first column of the table shows the average size of the predicates (in terms of clauses) representing the frame hierarchy. The next four columns of the table give time and space measurements for the Muse implementation:

- the time (in seconds) needed to `fcompile` the representation of the whole disease hierarchy into quick load (`.ql`) format,
- the time to load the `.ql` file,
- the memory used for loading (as displayed by the `load` built-in),
- the (wall-clock) time needed to run the `diagn1` benchmark on a single processor version of Muse.

[4] We avoided the use of the foreign interface, because of significant speed difference of its implementation in the two systems.

[5] The only major exception to this is in the overheads of the multiple binding scheme: in Aurora the binding arrays technique of the SRI model has about 25% time overhead, while the Muse copying approach has about 5% overhead on single processor execution.

The last column shows the single processor wall-clock execution time of Aurora on the `diagn1` benchmark.

As shown in row 1, the *single* predicate representation is characterised by very high load time as well as unacceptably high execution time. We therefore excluded this variant from our parallel experiments.

Row 2 shows the *many* predicates variant, where each frame is stored as a separate predicate. This representation has good execution time characteristics, but high storage requirements. It is interesting to note that Aurora is faster than Muse in this case. Further analysis shows that search in this representation inherently relies on meta-calls, and these in the Muse engine are about a magnitude slower than in the Aurora engine. This is because the Muse implementation of meta-calls uses additional Prolog code to cater for modularity and goal expansion, features that are not present in Aurora. To make the comparison more fair, we added to our evaluation a Muse variant, called *many**, which, instead of the `call/1` predicate, uses an internal system predicate (`prolog:call_module/2`), to avoid the overheads.

Rows 3, 4 and 5 refer to representations using various hash functions to partition the frame representation into groups: prefix of 3 characters, and hashing with moduluses 17 and 257. They have very similar development figures, except that the load time for the *hash 257* representation is about half the other two. The average predicate size seems to have a big influence on the load time: for the *single* variant, with huge predicate size, we get a very high load time, while for *hash 257* variant, with a low average predicate size, we get the fastest load time.

Regarding execution times, it is interesting to note that variant *prefix 3* is again faster on Aurora than on Muse. Searching the frame hierarchy in the *prefix 3* representation relies on both meta-calls and the `atom_chars/2` built-in predicate. It turns out that composition of atoms from their characters can be several times slower in the Muse engine than in Aurora, because of the different atom table structure[6]. To separate the issue of meta-calls and atom composition we introduced a variant of the search code for this representation, called *prefix 3**, which avoids meta-calls by the usual technique of a switch predicate[7].

Summing up, variants *many* and *hash 17* are the two fastest for both Aurora and Muse in the sequential execution of the `diagn1` benchmark. We noted the same tendency for the other benchmarks in our suite. In the next section we give timings for all benchmarks for selected representation variants.

We now briefly discuss the issue of development efficiency of Aurora. As regards storage requirements, Aurora figures vary only a few percent with respect to those shown for Muse in Table 9.1. However, Aurora compilation and load times are several magnitudes bigger than those for Muse, due to SICStus 0.6 scaling up very badly in this respect. For variants 1 and 2 several hours are needed to compile and load the disease database. For variants 3-5, compilation and load both take 700 to 1500 seconds each. This deficiency of the Aurora engine is in part offset by its ability to produce saved states, which is absent in Muse[8].

9.5.2 Parallel performance

In this section we discuss the parallel performance of various SNOMED search problems on both Aurora and Muse for 1 to 6 processors. The four benchmarks described earlier are used in the analysis. In the case of `diagn1`, we use a sequence of ten invocations of the benchmark, to increase measurement accuracy. We also include the arithmetic mean of the four benchmarks in the tables[9].

Each benchmark has been measured 15 times for each number of processors and the smallest wall-clock times were taken into account. Entries in the tables show the (wall-clock) execution time in seconds followed by the speedup figure with respect to the 1 processor case (in italics).

Table 9.2 shows the Aurora and Muse execution time of the benchmarks using the *many* representation, for

[6] The Muse (SICStus 3) atom table is a tree structure that allows fast comparison of atoms, at the expense of slower atom construction. In Aurora (SICStus 0.6) atoms are stored in a simple hash table, resulting in fast construction of atoms (at least of those which are entered in the table early enough). Note that in the presence of many atoms, hash conflicts can slow down this algorithm as well.

[7] Note that this technique cannot be applied to variant 2, because of the huge number of predicates to be invoked via a meta-call.

[8] Restoring a saved state in Aurora, with the disease database loaded, takes about 35 seconds, which is comparable to the load time of the `.ql` files in Muse.

[9] We are aware of the fact that taking the arithmetic mean of execution times assigns a bigger weight to benchmarks running longer. However, having multiplexed `diagn1` 10 times, the benchmarks are roughly of the same size, and so taking an arithmetic mean does not distort the figures too much. We have compared the harmonic mean of speedups (which puts an equal weight on each benchmark) with the speedups calculated from the arithmetic mean of execution times, and found that these are within 0.5% of each other. The advantage of using the arithmetic mean of times, rather than the harmonic mean of speedups, is that this way we get an overall time figure, with which we can compare the two systems, Aurora and Muse.

Goals	Processors			
	1	2	4	6
Aurora *(many)*				
`diagn1*10`	7.41	3.77 *(1.97)*	2.04 *(3.63)*	1.57 *(4.73)*
`diagn2`	4.36	2.19 *(1.99)*	1.13 *(3.85)*	0.81 *(5.41)*
`browse1`	6.71	3.40 *(1.97)*	1.73 *(3.89)*	1.18 *(5.67)*
`browse2`	10.07	5.07 *(1.99)*	2.58 *(3.91)*	1.75 *(5.74)*
arith. mean	7.13	3.61 *(1.98)*	1.87 *(3.82)*	1.33 *(5.38)*
Muse *(many*)*				
`diagn1*10`	4.42	2.27 *(1.95)*	1.24 *(3.56)*	0.89 *(4.97)*
`diagn2`	2.38	1.19 *(2.00)*	0.63 *(3.78)*	0.43 *(5.53)*
`browse1`	4.32	2.20 *(1.96)*	1.13 *(3.82)*	0.79 *(5.47)*
`browse2`	6.84	3.46 *(1.98)*	1.80 *(3.80)*	1.23 *(5.56)*
arith. mean	4.49	2.28 *(1.97)*	1.20 *(3.74)*	0.83 *(5.38)*

Table 9.2: Parallel performance of the "many" representation

1, 2, 4 and 6 processors. In the case of Muse the *many** variant was used, to avoid excessive overheads of meta-calls. In this representation Muse is about 60% faster than Aurora, on average. With 6 processors, Aurora speedups are somewhat lower on the diagnosis benchmarks while Muse has lower efficiency on the browse searches. Given the large amount of data to be searched, the overall speedup of about 5.4 for 6 processors is very good. Also, the 6 processor execution time of the largest searches is below 2 seconds, which is acceptable for interactive use.

Goals	Processors			
	1	2	4	6
Aurora				
`diagn1*10`	9.26	4.73 *(1.96)*	2.54 *(3.65)*	1.96 *(4.74)*
`diagn2`	4.79	2.44 *(1.97)*	1.23 *(3.88)*	0.88 *(5.43)*
`browse1`	7.02	3.55 *(1.98)*	1.81 *(3.88)*	1.23 *(5.72)*
`browse2`	10.32	5.22 *(1.98)*	2.64 *(3.92)*	1.79 *(5.75)*
arith. mean	7.85	3.98 *(1.97)*	2.05 *(3.82)*	1.47 *(5.36)*
Muse				
`diagn1*10`	10.75	5.63 *(1.91)*	3.08 *(3.49)*	2.84 *(3.79)*
`diagn2`	5.40	2.79 *(1.94)*	1.49 *(3.62)*	1.30 *(4.15)*
`browse1`	6.92	3.58 *(1.93)*	1.93 *(3.59)*	1.39 *(4.98)*
`browse2`	9.52	4.93 *(1.93)*	2.60 *(3.66)*	1.83 *(5.20)*
arith. mean	8.15	4.23 *(1.92)*	2.27 *(3.58)*	1.84 *(4.43)*

Table 9.3: Parallel performance of the "prefix 3*" representation

Table 9.3 shows the performance figures for the *prefix 3** representation. On a single processor Aurora is about 10% slower on average for this variant than for the *many* representation. In contrast, the average Muse execution time almost doubles. As discussed earlier, this is due to the difference in the implementation of the atom construction function in the underlying Prolog engines.

The multi-processor performance of Muse is much worse than that of Aurora: the gap in the average speedup increases with the number of processors. This highlights a further problem with atom construction: in both Aurora and Muse this operation is guarded by a single global lock, as adding a new atom to the table is

required to be an atomic operation. As the Muse atom construction operation is more expensive, it causes more contention for locks, and hence less efficient exploitation of parallelism. This is more apparent for the diagnosis benchmarks, which are dominated by the tree search.

In principle, locking at atom construction could be avoided, when this does not result in the creation of a new atom. This is actually the case in the frame representation discussed, as the atoms constructed are all names of existing predicates. Unsynchronised atom search, on the other hand makes creation of new atoms a more complex operation, and also care has to be taken to avoid problems of interference between atom search and the creation of new atoms (e.g. new atom creation may cause the system to extend the atom table at the same time when atom searches are done by other processors).

We plan to experiment with modifying the or-parallel systems discussed to avoid locking at atom construction, as this may seriously improve parallel efficiency of programs intensively using atom-handling operations.

Goals	Processors			
	1	2	4	6
Aurora				
`diagn1*10`	9.14	4.65 *(1.97)*	2.48 *(3.68)*	1.91 *(4.79)*
`diagn2`	4.75	2.40 *(1.98)*	1.23 *(3.88)*	0.86 *(5.50)*
`browse1`	6.92	3.51 *(1.97)*	1.78 *(3.89)*	1.22 *(5.69)*
`browse2`	10.02	5.09 *(1.97)*	2.58 *(3.88)*	1.76 *(5.69)*
arith. mean	7.71	3.91 *(1.97)*	2.02 *(3.82)*	1.44 *(5.36)*
Muse				
`diagn1*10`	5.06	2.60 *(1.95)*	1.42 *(3.56)*	1.03 *(4.91)*
`diagn2`	2.71	1.36 *(1.99)*	0.70 *(3.87)*	0.50 *(5.42)*
`browse1`	4.54	2.31 *(1.97)*	1.19 *(3.82)*	0.82 *(5.54)*
`browse2`	7.14	3.62 *(1.97)*	1.89 *(3.78)*	1.29 *(5.53)*
arith. mean	4.86	2.47 *(1.97)*	1.30 *(3.74)*	0.91 *(5.34)*

Table 9.4: Parallel performance of the "hash 17" representation

Table 9.4 shows the parallel performance figures for the *hash 17* representation. This variant is on average 8% slower than the one using the *many* representation. The speedups are roughly equal to those for the *many* variant.

We have made parallel performance measurements for the *hash 257* representation as well. We got data very similar to those in Table 9.4 with a very slight (1-3 %) overall reduction in both absolute speed and speedups achieved. The slight slow-down seems to be connected to the size of the predicates involved: in the *hash 17* version each frame attribute access involves a call of a 17-clause predicate and (on average) a 1100-clause one, while in the *hash 257* variant the predicates invoked have 257 and 72 clauses on average. We plan to further explore the reasons behind this behaviour in the future.

Processors	Variants					
	many	many*	prefix 3	prefix 3*	hash 17	hash 257
1	0.96	1.59	0.75	0.96	1.59	1.62
2	0.95	1.58	0.74	0.94	1.58	1.61
3	0.95	1.57	0.73	0.93	1.56	1.59
4	0.94	1.56	0.71	0.90	1.55	1.57
5	0.96	1.56	0.72	0.85	1.55	1.58
6	0.97	1.59	0.72	0.80	1.58	1.59

Table 9.5: Muse/Aurora speed ratios for different representations

As a final comparison of the two or-parallel systems examined, Table 9.5 shows the speed ratio of Muse and Aurora for the arithmetic mean of the four benchmarks, measured using six different frame representation techniques for 1 to 6 processors. It is quite interesting to see how little the figures in a single column vary: except for the *prefix 3** variant (which has a high contention for the atom table lock in Muse), the relative speed of Aurora and Muse changes less than 5% when the number of processors changes. This means that on the SNOMED benchmark suite Aurora and Muse achieve very similar speedups, although their single processor speed varies, depending on the representation chosen.

Columns with figures below 1 highlight features that became slower in Muse with respect to Aurora: the *many* representation uses meta-calls, *prefix 3* uses meta-calls and atom construction, while *prefix 3** uses only atom construction predicates. Interestingly, variant *prefix 3* does not show as much slow-down for Muse as the *prefix 3** version, as the overhead of meta-calls results in longer execution time and thus reduces the congestion for locks.

9.5.3 Summary

According to performance results discussed above, the *many* frame-representation has the fastest execution time, both for single and multiple processor execution, for Aurora as well as for Muse. Note, however, that to achieve this speed in Muse the *many** variant, with "doctored" meta-calls, had to be used.

A significant drawback of the *many* representation is its large storage requirement, which is over double the one for other variants. As the speed of the *hash 17* representation is only 8% slower for both systems, we have decided to use a hash-based frame-representation technique in the final version of the CUBIQ tool-set. We allow the modulus to be selected at the start-up of the tool-set, to allow fine tuning of applications.

9.6 Conclusions

Within the CUBIQ expert system tool-set we have implemented a frame-extension of Prolog. We have explored several techniques for frame representation, examining their ability to support parallel search in large frame hierarchies.

We have evaluated these techniques by implementing the SNOMED medical thesaurus using CUBIQ frames, and analysed the performance of searches within the SNOMED frame hierarchy using various representations. We have also examined the feasibility of implementing the SNOMED hierarchy in SICStus Prolog Objects.

For our parallel experiments we have used two or-parallel Prolog systems, Aurora and Muse, both based on (different versions of) SICStus Prolog.

We have shown that the 18,000 node SNOMED disease hierarchy can be efficiently represented in Prolog, using the general frame-extension of the CUBIQ tool-set. We have developed an implementation for CUBIQ frames, based on the `term_hash` SICStus predicate, with good time and space characteristics. We have shown that, for both Aurora and Muse, about 90% parallel efficiency can be achieved for six processors in complex searches of the SNOMED hierarchy.

Our experiments with various frame representations highlighted a number of interesting features in the Prolog implementations used. We have found some Prolog elements, such as meta-calls and the atom construction function, the implementation of which is much slower in the newest SICStus engine than in the older one. We have shown that the synchronisation done at atom construction hinders parallel execution of programs that use this function very often. We have also pointed to some implementation details that make parallel execution of object hierarchy searches in SICStus Prolog Objects infeasible.

We have compared the parallel behaviour of Muse and Aurora on SNOMED searches using different frame representation and search techniques. We have found that although their relative speed varies, the speedup figures of the two systems are very similar.

Our future plans include the modification of some critical parts of Aurora and Muse to avoid some of the problems highlighted by our experiments, such as unnecessary synchronisation at atom creation. We also hope that with the further development of the EMRM prototype we will be able to test the parallel behaviour of other, more complex search problems as well.

Acknowledgment

The authors are indebted to all their colleagues in the CUBIQ project and gratefully acknowledge the support of the European Union Copernicus programme, under project CP93-10979 'CUBIQ'.

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse Or-Parallel Prolog model and its performance. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776, Austin, 1990. ALP, MIT Press.

[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] David Dodson, Hugh Reeves, and Rob Scott. ICD-Edit: A server for $2\frac{3}{4}$-D interactive connection diagram graphics with Prolog clients. Technical report TCU/CS/1995/2, Department of Computer Science, City University, 1995. Poster presentation at GD'94, Princeton, New Jersey, October 1994.

[4] Zsuzsa Farkas, Péter Szeredi, and Gábor Umann. CUBIQ tool-set reference manual, version 4. Technical report, IQSOFT Ltd., Hungary, 1995.

[5] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[6] Katalin Molnár. Parallel Prolog with uncertainty handling. In *EUROPAR'95 Parallel Processing*, pages 691–694. Springer, 1995. Lecture Notes in Computer Science 966.

[7] D. J. Rothwell, R. A. Cote, J. P. Cordeau, and M. A. Boisvert. Developing a standard data structure for medical language — the SNOMED proposal. In *Proceedings of 17th Annual SCAMC, Washington*, 1993.

[8] SICS Programming Systems Group. Prolog Objects. In *SICStus Prolog User's Manual*, chapter 29, pages 275–307. Swedish Institute of Computer Science, June 1995.

[9] Péter Szeredi and Zsuzsa Farkas. Handling large knowledge bases in parallel Prolog, 1996. Presented at the Workshop on High Performance Logic Programming Systems, in conjunction with Eighth European Summer School in Logic, Language, and Information, Prague, August 1996.

[10] Gábor Umann, Robert B. Scott, David C. Dodson, Zsuzsa Farkas, Katalin Molnár, László Péter, and Péter Szeredi. Using graphical tools in the CUBIQ expert system tool-set. In *Proceedings of the Fourth International Conference on Practical Applications of Prolog*, pages 405–422, 1996.

[11] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

Chapter 10

Serving Multiple HTML Clients from a Prolog application[1]

Péter Szeredi, Katalin Molnár and Rob Scott[2]

IQSOFT Intelligent Software Ltd. H-1142 Teleki Blanka u. 15-17
Budapest, Hungary
{szeredi,molnark,scott}@iqsoft.hu

Abstract

The paper describes our experiences with transforming a medical expert system to a client-server architecture using an HTML interface.

We briefly present the expert system and describe the experiences of its transformation to the HTML-based user interface. We then focus on the issue of designing a single Prolog server capable of serving multiple client requests.

We present a solution based on an or-parallel Prolog system, Aurora. This approach allows the server to perform independent Prolog searches for each client, controlled interactively by the remote user.

Keywords: Expert systems, HTML, Client-server architectures, Prolog, Parallelism

10.1 Introduction

The EMRM (Electronic Medical Record Management) system prototype [1] has been developed in the CUBIQ Copernicus project, using SICStus Prolog [13] and its or-parallel extension, Aurora [9]. The original EMRM system uses a Tcl/Tk-based [11] forms interface for interacting with a single client. To widen the usability of the system, we are now developing an HTML-based interface to EMRM. This approach allows the users to access the system from a heterogeneous computer network (local network or Internet), with much smaller resource requirements on the local computers.

Our efforts are an example of a general trend in the AI community, to use the extended visibility that WWW provides to make applications available through the Internet. In order to interact with end-users through the WWW, such applications normally rely on socket-based inter-process communication features. Most commercial Prolog systems (ALS, Quintus, SICStus, etc.) already have such features implemented. Tools

[1] This paper has appeared in the proceedings of the Workshop on Logic Programming Tools for INTERNET Applications [15]

[2] Part of the work reported here has been carried out while the author was at the Computer Science Department, City University, Northampton Square, London EC1V 0HB, UK

have also been developed for helping the communication with the end-user, such as the CGI handler interface of [5] and the support functions of html.pl [4] for generating HTML documents from Prolog.

One of the major issues that, we believe, have not been addressed so far is the problem of a single Prolog program acting as a server for multiple WWW clients. This issue is important as AI applications are normally large and slow to start up, so having a separate copy of the application running for each request may not be a viable solution.

Handling multiple clients means that multiple threads of control have to be handled in a single Prolog program. Having explored several approaches to this problem, in the case of EMRM, we found that achieving this goal in a traditional sequential Prolog implementation requires serious changes in the formulation of expert system rules, which results in losing the clarity of the original knowledge base. On the other hand we found that an or-parallel Prolog implementation can be used to provide the functionality of serving multiple clients, while keeping the knowledge base intact and preserving the declarative style of knowledge representation.

In the following we briefly outline the main functions of the EMRM and describe the major steps in transforming its user interface into HTML. We then discuss the problems stemming from the asynchronous nature of HTML communication, for both the single client and multiple client versions. Finally we outline a solution for asynchronous handling of multiple clients using an or-parallel Prolog implementation.

10.2 An overview of EMRM

EMRM is designed to help physicians to get as much relevant information about their patient as possible before meeting the patient personally. While waiting for the doctor the patient can give information to a medical assistant who is supported by EMRM. Some of the questions are generated on the basis of the information already collected about the patient, and the medical knowledge incorporated in the knowledge base of EMRM. The answers of the patient are stored using the relevant medical terms of SNOMED [12], a medical thesaurus of over 40,000 terms. The physician gets the collected data and the possible diagnoses suggested by EMRM before meeting the patient. He can then collect further information by examining the patient and checking for further symptoms. These data, the diagnoses and the further diagnostic and therapeutic steps, are entered into the EMRM system by the physician and again stored as relations built from SNOMED terms.

The Prolog implementation of EMRM thus consists of the following main parts:

- dialogue management
- medical rule-base
- SNOMED thesaurus

EMRM is a relatively large program (requiring about 20 Mb memory in the present prototype phase).

10.3 EMRM with a HTML user interface

In the process of transforming EMRM to a HTML interface we first implemented an HTML version of the SNOMED browser, an important component that allows the user to find the appropriate medical term by a combination of tree traversal and text search [10]. We then continued the transformation process for the main dialogue of EMRM, including data entry and the display of the results.

The HTML version of the EMRM server consists of the Prolog program, a WWW server and and a small CGI script that mediates between the Prolog program and the WWW server. The Prolog program is running as a separate server process on the WWW server computer. Because of the large program size and relatively slow startup time it is reasonable to have the Prolog program running all the time, rather than to start it up separately for each request.

In the Prolog program there is a main loop waiting for client requests on a socket from the WWW-server. The user interface of the Prolog program is based on "lazy" querying techniques. i.e. the user is asked for some information only when the deduction process requires this. The interaction with the user is done

through forms that are dynamic HTML pages. Whenever the user is asked for some information through such a form, the system sends out the dynamic HTML page and waits for the answer. The evaluation of the Prolog program is then suspended until an answer from the client process arrives. Such a low level wait-loop is part of any program code that implements forms. By the nature of the application such form requests are embedded within the Prolog search tree generated by the medical rule-base.

The WWW server communicates with the Prolog program through the CGI script, that is invoked whenever there is some communication from the user interface side.

10.4 Problems with single client

The first problem, due to the asynchronous nature of HTML communication, already arises in the case of a single client. An HTML page does not completely disappear after the user has answered the query on the page, as it normally remains available through the *history* function of the browser (Netscape, Internet Explorer, etc.). Thus, all the previous queries are there, and in principle the user can go back at any time, change and re-submit some of the earlier answers.

In our present implementation we forbid such re-submission requests. In principle, the backtracking ability of Prolog could be used to interpret such actions as requests to forget all information gathered since the original of the re-submitted HTML page, and to restart the search with a new answer to the query on that page.

This can be accomplished in a way similar to the implementation of the retry function available in most Prolog debuggers. This function relies on a choice point (such as created by `repeat/0`) being placed in front of each retry-point. The retry operation is then performed by doing a far-reaching cut (ancestor-cut), pruning all choice points up to the chosen retry-point and then failing the computation. This unwinds the Prolog stacks up to the required point and, because of the `repeat`, restarts the computation there.

For this approach to work, the program should not be altering the Prolog dynamic database, so that execution of the retried goal is restarted in the same setting as the original execution. As a possible exception to this rule, earlier answers could be stored in the database, and used as the default selections in the repeated queries.

10.5 Serving multiple clients

The second, more serious problem arises when dealing with several clients. E.g. if we have two clients, it should be possible to process their answers in some interleaved way, following the relative timing of their answers. In other words, we can have two clients, in different stages of evaluation, both waiting for some user answer, and we should be able to continue the execution with whichever client answers earlier.

The simplest way to achieve this coroutining behaviour is by requiring that the program in question is formulated as a set of separate actions to be carried out in response to the arrival of an answer. Although such a set-up is easy to implement, in most cases it completely destroys the logic of the program. This is not only because communication between phases has to be done through the Prolog dynamic database (rather than logic variables), but more importantly because the original logical structure of the rule-base cannot be preserved.

We have considered several options for implementing the coroutining execution of an existing Prolog program, without requiring its reformulation:

- developing a coroutining interpreter,
- using existing coroutining features (such as freeze), possibly combined with some compilation scheme,
- using primitives for handling continuations as first class objects (as e.g. in the early implementation of CS Prolog [7]).

A common drawback of these schemes is that they work properly only with deterministic code. As they all rely on chronological backtracking in a single stack regime, backtracking in one of the coroutined branches will lead to (unnecessary) backtracking over the interleaved execution steps of other branches.

As EMRM heavily relies on backtracking, we need a solution that allows independent backtracking in the coroutined branches.

A natural approach is to consider parallel Prolog systems. There are several approaches in this area which support independent exploration of multiple searches: or-parallel systems, such as Aurora; systems supporting independent and-parallelism, such as the &-Prolog system [8]; and systems with explicit control of parallelism, such as CSR Prolog [7], and BinProlog extensions [3].

As one of goals of the CUBIQ project was to examine the usability of or-parallel Prolog systems in expert system applications, it was natural for us to explore whether Aurora can be used to support multi-client EMRM execution.

10.6 Using an or-parallel Prolog as a multi-client server

Aurora [9] is an or-parallel implementation of full Prolog based on SICStus 0.6. In Aurora a number of *workers* (i.e. processes, normally running on separate processors of a multiprocessor computer) work together on exploring the search tree corresponding to the Prolog program. Aurora preserves sequential Prolog semantics, e.g. the side-effect predicates, such as the ones for input-output and dynamic database handling, are executed in exactly the same left-to-right order as in a sequential Prolog.

This section outlines how Aurora can be used to support the execution of multiple independent copies of a Prolog program, which interact with remote users through the WWW. It is crucial for this purpose that Aurora supports non-synchronised execution of side-effect predicates, by providing so called cavalier variants, written as

`cavalier(`*Pred*`)` e.g. `cavalier(write(foo))`.

Cavalier predicates are executed immediately, without any synchronisation with other branches. A typical usage of such predicates is to display tracing information on how the parallel execution proceeds.

Cavalier predicates are executed atomically, i.e. if two competing branches reach a side-effect predicate affecting the same resource[3] simultaneously, then these predicates will be executed in some arbitrary order, one after the other. For example, if a sequence of terms is displayed by a cavalier `format` predicate, this sequence is guaranteed not to be intermixed with output coming from other branches of the Prolog search tree.

Aurora also allows the user to control which predicates can be executed in parallel, by appropriate declarations.

We now first present a general skeleton of a multi-client Prolog server (Figure 10.1) and then describe how the process spawning and communication primitives of the skeleton can be implemented in Aurora.

```
loop(Socket) :-
   repeat,
   next_event(Socket, Event),
   process_event(Event),
   fail.

process_event(session(S)) :-
   spawn(run_session(S)).
process_event(answer(Req, Answer)) :-
   out(Req, Answer).

query(Req, Answer) :-
   in(Req, Answer).
```

Figure 10.1: The main loop of the multi-client execution scheme

[3] e.g. the same dynamic predicate, or the same output stream

The predicate `loop` in Figure 10.1 implements the (non-terminating) main loop of the multi-client server. The argument of `loop` represents the socket used for network communication. This predicate first waits for the next network event (in `next_event`), and then processes the event through `process_event`. An event can be a request to start a new session, represented by a Prolog term of form `session(S)`. This is processed by spawning a fresh copy of the Prolog server program (`run_session`).

When, applying the lazy querying technique, further input from the remote client is needed within a session, the Prolog code for the session has to call the `query` predicate of the above skeleton[4]. A query has a unique request identifier (`Req`) as its input argument, and when the answer arrives it instantiates its `Answer` output argument. The answer to the query is detected as an event of form `answer(Req, Answer)` in the main loop. Processing of this event requires communicating the answer to the session waiting for it, this communication is done through two procedures named following the Linda convention [6] as `out(Req, Answer)` (on the sender side, in the main loop) and `in(Req, Answer)` (at receiving end, in the Prolog session).

We now proceed to show how the primitives `spawn`, `in` and `out` can be implemented in Aurora. As a first step let us assume that all of the user program (`run_session`) is declared to be sequential, i.e. parallelism is only used for implementing multiple execution of Prolog sessions.

```
:- parallel spawn/1.

spawn(Goal) :-
   (
       true
   ;
       call(Goal) -> fail
   ).

in(Req, Answer) :-
   repeat,
   cavalier(retract(request(Req,Answer))),
   !.

out(Req, Answer) :-
   cavalier(assert(request(Req,Answer))).
```

Figure 10.2: Simple implementation of process handling primitives

Figure 10.2 shows a simple implementation of the communication primitives. Spawning is achieved by simply opening a parallel choice with two alternatives: the first one is empty, thus returns to the callee immediately, while the second one calls the goal to be spawned. The Aurora parallel scheduler ensures that such a new parallel alternative is executed by an idle worker, if one exists. Note that this approach does not require the forking of a new process for each spawn operation: Aurora uses a fixed number of workers (processes), which are scheduled to execute tasks as they arrive (and sleep if there are no tasks to execute).

In this implementation, the communication between the main loop and the spawned processes is implemented using a dynamic predicate, `request/2`, as an application of general techniques described in [14]. The `in/2` predicate spins in a busy waiting loop until the `out/2` asserts the requested answer. The assert and retract predicates have to be cavalier, so that they are executed irrespectively of their position in the Prolog or-tree.

The solution for process communication presented in Figure 10.2 has several drawbacks. First, workers wait in a busy loop, thus wasting computing resources. This problem could be overcome by inserting a Unix sleep in the loop, thus making the processor available for other computations. This solution, however, still does not make the or-parallel scheduler aware of the fact that the execution branch in question is suspended. As the Aurora system can be started up with a fixed number of workers, say N, this means that at most $N-1$ sessions can be alive at any moment (one worker is executing the main loop).

As a fairly recent development, new primitives have been introduced in Aurora for user-controlled suspension

[4] The call to `query` is normally preceded by sending an HTML page to be filled in to the remote client. Communication in this direction is fairly straightforward and is not discussed here.

and resumption of execution branches [2]:

- `force_suspend(L)`

  Forces the current branch to suspend, and assigns it the label L. Labels currently can only be integers.

- `resume_forced_suspend(L)`

  This marks the suspended branch labeled L as resumable and continues with the current branch. The scheduler may schedule a worker to restart the suspended branch any time after this resumption operation has been executed.

The new features make it possible to avoid the busy waiting in process communication.

```
in(Req, Answer) :-
   force_suspend(Req),
   cavalier(retract(request(Req,Answer))).

out(Req, Answer) :-
   cavalier(assert(request(Req,Answer))),
   resume_forced_suspend(Req).
```

Figure 10.3: Process communication based on user controlled suspension

Figure 10.3 shows the implementation of `in` and `out` with the new primitives. This approach does notify the Aurora scheduler about the branch becoming suspended and makes it possible for the scheduler to use the worker for executing code at other parts of the search tree. Consequently, it allows Aurora to be run with two workers only and still serve any number of requests, with interleaved execution. Since one of the workers is waiting for network input most of the time, we believe that such a two-worker Aurora configuration can be safely run on a mono-processor computer as well.

On the other hand, if a multiprocessor is available as a server, there is no need to forbid the exploitation of parallelism in the Prolog programs run. The Aurora scheduler will ensure that parallelism is exploited both between the independent threads of execution and within such threads.

10.7 Present status and future work

We have implemented the single-client version of EMRM. Figure 10.4 presents a Web page with the start-up query of EMRM. We have designed the support for multi-client execution of Prolog servers and tested the first version of process communication primitives on simple examples.

We plan to continue the development of the multi-client version of EMRM. We also hope to compare our method with approaches based on other parallel Prolog systems, such as &-Prolog and parallel BinProlog.

10.8 Conclusion

We have outlined EMRM, a Prolog application implementing a medical record management expert system. We have described our work on transforming the user interface of EMRM to apply a WWW browser communicating with the Prolog program through HTML forms and HTML pages.

We have outlined some techniques for interleaved execution of multiple copies of a server application in Prolog. We have pointed out that most of these are not capable of supporting the proper interleaving of multiple backtracking searches.

We have presented our design for a single Prolog server, based on an or-parallel implementation Aurora, supporting multiple clients with full support for independent Prolog search. The main advantage of this approach over running a separate copy of the application for each client is the avoidance of lengthy start-up

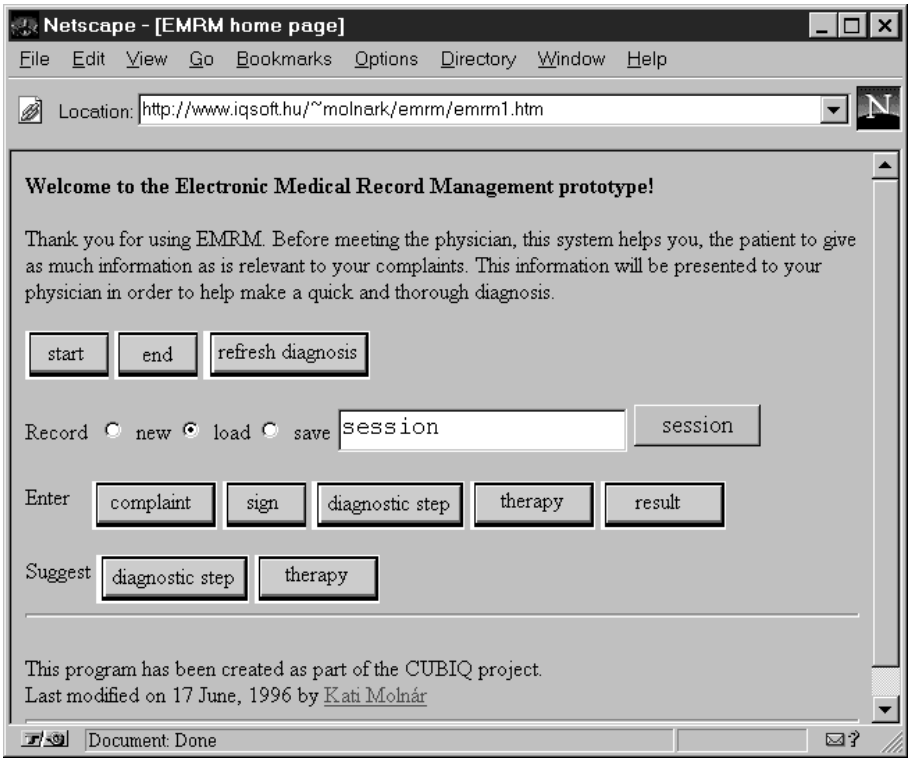

Figure 10.4: A snapshot of the EMRM opening page

time and significant reduction in memory requirements. As a further advantage the single server approach allows easy communication between the program instances serving the different clients, which may be useful e.g. for caching certain common results, collecting statistics, etc. An important aspect is that the knowledge base does not need to be changed and the commonly used lazy querying techniques can be safely applied.

We hope to complete the development of the multi-client version of EMRM in the near future. We believe this server will be able to serve several clients simultaneously, at a reasonable speed and with reasonable resources.

Acknowledgement

The authors are indebted to all their colleagues in the CUBIQ project and gratefully acknowledge the support of the European Union Copernicus programme, under project CP93-10979 'CUBIQ'.

References

[1] László Balkányi, Zsuzsa Farkas, and Katalin Molnár. EMRM Electronic Medical Record Management System. CUBIQ Copernicus project deliverable report, IQSOFT Ltd., Hungary, 1995.

[2] Tony Beaumont, David H. D. Warren, and Péter Szeredi. Improving Aurora scheduling. CUBIQ Copernicus project deliverable report, University of Bristol and IQSOFT Ltd., 1995.

[3] Koen de Bosschere and Paul Tarau. Blackboard-based extensions for parallel programming in BinProlog. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, page 664, Vancouver, Canada, 1993. The MIT Press.

[4] D. Cabeza and M. Hermenegildo. html.pl: A simple HTML package for Prolog and CLP systems. Technical report, Computer Science Department, Technical University of Madrid, 1996.

[5] B Carpenter. A Prolog-based CGI handler, 1996. `http://macduff.andrew.cmu.edu/cgparser/prolog-cgi.html`.

[6] N. Carreiro and D. Gelernter. Linda in context. *Comm. of the ACM*, 32(4), 1989.

[7] Iván Futó. Prolog with communicating processes: From T-Prolog to CSR-Prolog. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 3–17, Budapest, Hungary, 1993. The MIT Press.

[8] M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268, Jerusalem, 1990. The MIT Press.

[9] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David 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.

[10] Katalin Molnár, Robert B. Scott, and Zsuzsa Farkas. HTML as a user interface for a (Prolog) program. In *Poster Proceedings of the 4th World Wide Web Conference*, 1995.

[11] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[12] D. J. Rothwell, R. A. Cote, J. P. Cordeau, and M. A. Boisvert. Developing a standard data structure for medical language — the SNOMED proposal. In *Proceedings of 17th Annual SCAMC, Washington*, 1993.

[13] SICS Programming Systems Group. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, June 1995.

[14] Péter Szeredi. Using dynamic predicates in an or-parallel Prolog system. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Logic Programming Symposium*, pages 355–371. The MIT Press, October 1991.

[15] Péter Szeredi, Katalin Molnár, and Rob Scott. Serving multiple HTML clients from a Prolog application. In Paul Tarau, Andrew Davison, Koen de Bosschere, and Manuel Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications, in conjunction with JICSLP'96, Bonn, Germany*, pages 81–90. COMPULOG-NET, September 1996.

Conclusions

This thesis describes work on the Aurora or-parallel Prolog system. To conclude, we first summarise the contributions of the author, and then give a brief evaluation of the problems encountered, their solutions, and the significance of the results.

Contributions

The following is a summary of the main results achieved by the author of this thesis.

Implementation: I developed a profiling technique for Aurora and carried out detailed performance analysis in the early stages of the project. This profiling technique has been used throughout the project and significantly helped subsequent design decisions. I was the principal designer of the engine-scheduler interface, which enabled the development of multiple scheduler and engine components. I designed the basic Bristol scheduler which later evolved to be the main scheduler used in Aurora.

Extensions: I developed two language extension proposals to support advanced search techniques in or-parallel Prolog systems: for synchronisation of dynamic predicate updates and for advanced optimisation search involving branch-and-bound and alpha-beta pruning. I developed a prototype implementation for both extensions in Aurora and carried out case studies to prove their usefulness.

Applications: I was a principal contributor to several application projects in diverse areas: computational molecular biology, hierarchical knowledge bases and WWW-servers. These applications prove the viability of Aurora and of or-parallel Prolog systems in general.

Evaluation

The conclusion of the Aurora overview paper (Chapter 2, Section 2.7) gives a summary of how the Aurora development team viewed its results as of 1989. We now try to reiterate the main issues raised there and give an up-to-date evaluation of the Aurora project.

We already stated in 1989 that Aurora demonstrated the feasibility of the SRI model as a means for transforming an efficient sequential Prolog to an *or-parallel engine*. Since then, the main improvements on the engine side were linked to the development of the second generation of Aurora with SICStus 0.6 as its core. This Aurora system contains the new engine-scheduler interface, as described in Chapter 5.

Today, the SICStus 0.6-based Aurora engine is fairly outdated. Re-building the engine on top of an up-to-date Prolog, such as the current SICStus3 implementation, does not pose any conceptual problems, but it does require substantial effort to be invested.

The 1989 Aurora paper lists several outstanding issues in *scheduling*, which have been solved since then, such as better scheduling heuristics and handling of speculative work. The Bristol scheduler, described in Chapter 4, applied a new, "dispatching on bottom-most" heuristics, which resulted in coarser task granularity and reduced task switching overheads. Further improvements to Bristol scheduler [1] provided support for better scheduling of speculative work. Also, a new scheduler, called Dharma, was developed [8], applying the so called "branch level scheduling" approach, which also gives good results on both speculative and non-speculative work.

The main problems of supporting the *full Prolog language* in an or-parallel setup were solved by 1989. However, to make the system usable in practice, several further issues had to be tackled: selecting the precise set of asynchronous built-in predicates; defining and implementing the dynamic database update semantics for the asynchronous case; solving the problems of parallel file input-output; providing immediate (as opposed to post-mortem) parallel tracing facilities. Having solved these problems [2], Aurora became the first full-fledged Prolog system capable of exploiting or-parallelism in arbitrary Prolog programs.

The 1989 paper lists three major *applications* of Aurora, Since then several further applications were successfully ported to Aurora, including the ones described in this thesis (Chapters 8–10) as well as others, e.g. [4, 7, 3]. Exploration of further application areas is opened up by work described in the language extension part of the thesis (Chapters 6–7).

Regarding the *multiprocessor* hardware, an important new development was the porting of Aurora to the BBN GP1000 and TC2000 machines, with non-uniform memory access (NUMA) architecture [5]. While the traditional multiprocessors scale up to about 30 CPUs, the NUMA machines can have a much larger number of processors. Aurora has shown almost linear speedups for programs with large search trees, including the molecular biology application of Chapter 8. A related development was the implementation of an emulator for the Data Diffusion Machine (DDM) virtual shared memory architecture [10] on transputer networks [6]. Aurora was ported to the DDM emulator and promising speedups were obtained.

As discussed in the 1989 paper, the biggest obstacle in obtaining truly competitive *bottom-line performance* is still the relatively high cost of multiprocessors. Machines with a high number of processors are still fairly expensive, but personal computers with 2 to 8 processors are becoming relatively cheap. We believe that such low-cost multiprocessor PCs, running a parallel Prolog implementation such as Aurora, will become cost-effective tools for solving search problems.

The main insights gained from the development of Aurora are the following. First, the decomposition of Aurora into scheduler and engine components was crucial in the development process. The strict engine-scheduler interface made it possible to experiment with different scheduling strategies and to re-use an Aurora scheduler in the Andorra-I implementation. Second, the problems of scheduling dominated the Aurora development. Five schedulers were developed with different scheduling principles and different underlying data structures. The scheduling algorithms became more and more concerned with exploiting parallelism in "difficult" cases such as very fine-grained parallelism, or speculative work. Third, Prolog, in spite of its declarative roots is still very much a sequential language. The Prolog community seems to prefer to think sequentially, e.g. the Prolog standard insists on all-solution predicates, such as bagof, returning the list of solutions in the sequential order. Observing such a restriction implies a significant overhead on parallel execution, which is unnecessary in a lot of cases. A positive example in this respect is the Mercury language[9], a new fully declarative logic programming language, the semantics of which does not contain any restrictions on execution order.

We believe that work on Aurora had a significant impact on research in parallel logic programming. Aurora has proved that it is feasible to support the full Prolog language in an or-parallel implementation. Aurora work included substantial research on scheduling or-parallelism, which can be used in other parallel implementations of logic programming. Aurora served as a basis for the Andorra-I system supporting both or- and and-parallelism. Aurora has proved that exploiting parallelism implicitly, without programmer intervention, is viable and can lead to substantial speedups in real-life applications.

References

[1] Tony Beaumont and David H. D. Warren. Scheduling Speculative Work in Or-parallel Prolog Systems. In *Logic Programming: Proceedings of the 10th International Conference*. MIT Press, 1993.

[2] 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.

[3] K. Eshghi and C. Preist. Model-based diagnosis applied to a real problem. Technical Report HPL-91-115, Hewlett Packard Laboratories, Bristol, UK, 1991.

[4] Feliks Kluźniak. Developing applications for Aurora. Technical Report TR-90-17, University of Bristol, Computer Science Department, August 1990.

[5] Shyam Mudambi. Performances of aurora on NUMA machines. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 793–806, Paris, France, 1991. The MIT Press.

[6] Henk L. Muller, Paul W. A. Stallard, and David H. D. Warren. The Data Diffusion Machine with a scalable point-to-point network. Technical Report CSTR-93-17, University of Bristol, October 1993.

[7] C.J. Rawlings, W.R.T. Taylor, J. Nyakairu, J. Fox, and M.J.E. Sternberg. Using Prolog to represent and reason about protein structure. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 536–543. Springer-Verlag, 1986.

[8] Raéd Yousef Sindaha. Branch-level scheduling in Aurora: The Dharma scheduler. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 403–419, Vancouver, Canada, 1993. The MIT Press.

[9] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[10] David H. D. Warren and Seif Haridi. Data Diffusion Machine–a scalable shared virtual memory multiprocessor. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.