SICS/R-89/8907

The Aurora Or-Parallel Prolog System

by
Ewing Lusk, David H.D. Warren, Seif Haridi,
et al.

SICS research report
R89007
ISSN 0283-3638

The Aurora Or-Parallel Prolog System

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

David H. D. Warren
Alan Calderwood
Péter Szeredi†
Bristol‡

Seif Haridi
Per Brand
Mats Carlsson
Andrzej Ciepielewski
Bogumil Hausman
SICS§

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.

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.

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

†*On leave from* SZKI, Donati u. 35-45, Budapest, Hungary

‡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.

§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 Background

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

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, 22]. 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, 30]. 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.

2.4 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 [26, 17, 29, 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 [27]. 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.

2.5 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.

3 Design

Aurora is based on the SRI model, and most of the design decisions are as described in an earlier paper [27]. In this section, we summarise the main features of the design, emphasising those aspects which are not covered in the earlier paper.

3.1 The Basic SRI Model

In the SRI model, a group of **workers**[1] 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.

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 [26, 12, 17], is that it imposes minimal overhead on a worker while it is working.

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

[1] 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.

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).

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

[27], 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).

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 [27]. 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.

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.

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].

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].

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.

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).

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.

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.

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.

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.

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[2], and returns control to the scheduler. It is up to the scheduler to decide when the suspended work may be resumed.

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

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.

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`.

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.

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 [27], 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 [23], 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.

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.

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 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.

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].

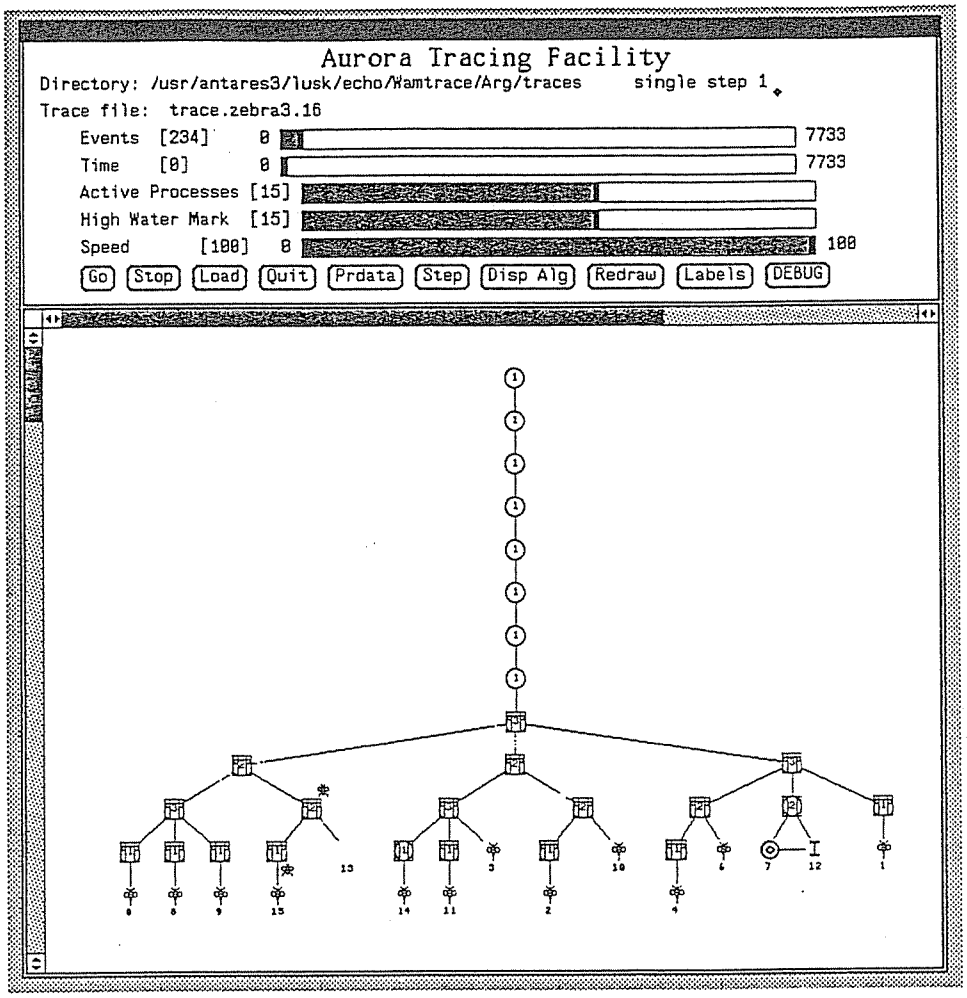

Figure 1: Snapshot of the graphical tracing facility

In Figure 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 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.

5 Experimental Results

In Tables 1, 2 and 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. 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[3]; `parse5`, the natural language parsing part of the same Chat-80 query.

Table 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 5 we present some sample profiling data obtained from running the same benchmarks on an instrumented version [23] 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

[3] "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?"

Benchmark	Processors 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 1: Times (in seconds) and speedups for Aurora, Manchester scheduler

Benchmark	Processors 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: Times (in seconds) and speedup for Aurora, Argonne scheduler

Benchmark	Processors 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 3: Times (in seconds) and speedup for Aurora, Wavefront scheduler

Benchmark	TIME (sec) Aurora 0.0 Symmetry	RELATIVE SPEED Sicstus 0.3 Symmetry	Sicstus 0.3 SUN 3/75	Quintus 2.4 SUN 3/75	Sicstus 0.6 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 4: Comparing speed of other Prolog implementations with Aurora

Benchmark	TOTAL TIME (sec)	TOTAL CALLS	TOTAL TASKS	CALLS /TASK	EXECUTION OVERHEADS	TASK SWITCHING OVERHEADS	IDLE TIME	LOCKING TIME	MIGRATION 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 5: Profile data for Aurora, Manchester scheduler, 20 processors

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 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 rela-

tively 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.

Remarkably similar speedups on the same benchmarks have been obtained for ECRC's PEP-Sys model [21]. 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.

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).

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"*.

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.

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 [20]. 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.

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, 30]. The work has also inspired ideas for a novel architecture supporting shared virtual memory, called the data diffusion machine [28]. We believe Aurora can contribute generally to the study of parallelism in logic programming languages.

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

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

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

[23] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. March 1989. To appear in the proceedings of the North American Conference on Logic Programming, 1989.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.