PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

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
Peter Szeredi†
Manchester‡

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, an existing, portable, state-of-the-art, 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 preliminary experimental results. We conclude with our plans for the continued development of the system and an outline of future research directions.

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.

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, evaluating, and refining one (or more) parallel execution models. The intention is to evaluate the models not just on the present hardware, but with a view to 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 in-

*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 Manchester, Manchester M13 9PL, U.K. *Now at:* Department of Computer Science, University of Bristol, Bristol BS8 1TR, U.K.

§Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden

cludes Prolog as a true subset.

The main purpose of this paper is to describe the issues that must be confronted in or-parallel Prolog implementation and detail the decisions and compromises made in Aurora. We include benchmark tests that point out the strengths and weaknesses of some of these decisions. We conclude by describing some 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.

2.1 Sequential Prolog Implementations

Prolog implementation entered a new era when the first compiler was introduced, for the DEC-10 [21]. 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) [22]. 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 parallelizing 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 utilizing an existing high-quality implementation as its foundation. We use the Sicstus implementation [7], 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 Multiprocessors

It is only in the last two 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. They are relatively inexpensive and provide a standard system environment (UNIX^{TM}) thus 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 specialized 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 determine whether it is feasible to propose logic programming as the vehicle for exploiting parallelism on these machines.

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) [14, 20]. These languages exploit **dependent** and-parallelism, in which there may be be dependencies between and-parallel goals. Other work [11, 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 te exploit transparently than and-parallelism. However, none of these reasons precludes integrating and-parallelism at a later stage, and indeed this is our ultimate intention.

- **Generality.** It is relatively straightforward to exploit or-parallelism without restricting the power of our 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 that 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 [23, 17, 26, 1, 10], incorporating different binding schemes.

An early binding scheme was that of the SRI model, first suggested informally by Warren in 1983 and subsequently refined [24]. The early form of this model partly influenced Lusk and Overbeek in the design of the pioneering system, ANL-WAM [13], 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 [8, 9, 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 of representatives of several groups interested in various aspects of parallelism in logic programming was held. It was agreed that there would be a core project, open to participation by anyone, and also that anyone whose research interests could benefit by completion of the core project was welcome to stay in close contact. Over the next year the core project came to be the Aurora system described in this paper, with Argonne National Laboratory, the University of Manchester, and the Swedish Institute of Computer Science as the implementors. 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. Others who attended these gatherings were representatives 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 concerning parallelism in logic programming.

3 DESIGN

Aurora is based on the SRI model, and most of the design decisions are as described in an earlier paper [24]. 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 destroyed) during the course of the exploration. Thus the first worker to enter a branch constructs it, and the last worker to leave a branch destroys it. The actions of constructing and destroying 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.

[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.

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 [23, 13, 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 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).

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.

The present Aurora implementation does not perform **straightening** or **promotion of bindings**, two possible memory management optimisations mentioned in the earlier paper on the SRI model [24].

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 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 [24]. 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 consistent. 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 processes, 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. 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. It is also intended to implement "cavalier" versions of certain predicates, which will 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 choice points 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 [4].

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 [5]. 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 version 0.3 [7], a relatively complete Prolog system implemented in C, which has been ported to a wide range of Unix machines. It 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 less than 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 indirect through 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, and returns control to the scheduler. It is up to the scheduler to decide when the suspended work may be resumed.

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 I/O. I/O is probably best 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 three quite distinct schedulers have been implemented (an early interim solution and two more recent and more complete solutions). These are described below. Others are being developed [3].

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 process would explore next. It was anticipated that this global lock would represent a bottleneck as machines with more processors become available. Both of the later schedulers use a more local scheme for assigning available work to available workers.

The two current schedulers are very similar in their level of completeness, both handling cut, commit and sequential side effects predicates correctly. Moreover, although they have rather different ways of implementing their responsibilities, they do share a number of strategy decisions. Both 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. Both schedulers attempt to maintain one, live, shareable node on their current branch, irrespective of whether any other worker is currently idle. If a cut or side effects predicate cannot be executed due to its not being on the leftmost branch in the appropriate subtree then both schedulers suspend that work, freeing the worker to look for another task. Neither scheduler currently concerns itself with speculative work, rather all work is regarded as being equally worthwhile.

4.2.1 The Manchester Scheduler

The aim of the Manchester scheduler [6] 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. 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 examined 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.

4.2.2 The Argonne Scheduler

The philosophy of the Argonne scheduler [4] 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.

4.3 The Underlying Multiprocessing Techniques

Aurora is based on the concept of a multiprocessor machine with a shared virtual address space. Although a number of vendors provide such machines, there is no portable standard by which process creation and synchronization of access to shared data structures are carried out. Portability of the system is achieved by using the approach described in [2] for writing portable parallel programs in C. A macro package, with definitions of a few basic operations customized for each multiprocessor, establishes a uniform syntax for creation of processes, management of shared memory, and accessing locks. The most commonly used technique for avoiding deadlock is a standard allocation pattern for the locks in the nodes of the tree, in which any process needing to lock two nodes (during traversal, for example) must obtain the lock for the upper node first.

4.4 Other Tools

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 package for replaying those events on a Sun workstation to show pictorially how the workers explore the search tree [12]. This tool has been extremely useful for investigating the behavior of the different schedulers. The Aurora system has also been instrumented to gather and analyse various salient statistics (see below).

5 EXPERIMENTAL RESULTS

In Table 1, we present some performance data for Aurora. The benchmarks considered are `8-queens2`, a naïve (generate and test) version of the 8 Queens problem from

Table 1:

TIMES and SPEED COMPARISONS for Aurora on Encore Multimax								
	TIME(sec)	SPEEDUP/RELATIVE SPEED						
	Aurora with N workers on Encore					Quintus Sun3/50	Sicstus Sun3/50	Sicstus Encore
Example	1	2	4	8	16			
8-queens2 (A)	53.77	1.98	3.89	7.53	12.4	4.85	1.99	1.29
salt-must2 (A)	2.25	2.15	4.11	8.02	14.2	4.25	2.01	1.20
tina (A)	40.84	1.97	3.84	7.22	11.3	4.29	1.89	1.26
db5 *10 (M)	7.24	1.88	3.38	5.62	6.35	4.44	1.98	1.14
parse5 (M)	11.35	1.87	3.42	5.28	5.83	4.89	2.22	1.25

Table 2:

STATISTICS for Aurora with 8 workers on Sequent Symmetry, Manchester scheduler										
	TOTAL			PER TASK						
	Time					Back-	WAM	Public	Moves	Move
Example	(sec)	Calls	Tasks	Calls	Nodes	tracks	binds	nodes	down	binds
8-queens2	6.46	167207	377	443	331	466	202	1.2	5.1	0.9
salt-must2	0.29	13574	107	127	60	44	28	0.7	1.9	0.6
tina	5.00	161365	1329	121	46	21	79	1.5	2.5	0.6
db5 *10	1.16	55450	1121	49	10	22	7	0.6	1.7	0.5
parse5	1.98	39096	1141	34	14	18	33	0.9	8.3	32

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 query[2]; parse5, the natural language parsing part of the same Chat-80 query. We show times and speedups for Aurora running on an Encore Multimax (the new Multimax 320 with APC) with different numbers of workers, and, for comparison, the relative speed of Quintus Prolog (on a Sun 3/50) and Sicstus Prolog (on a Sun 3/50 and Encore Multimax). Examples marked (A) were run with the Argonne scheduler and those marked (M) with the Manchester scheduler, the selection being determined by which scheduler currently gives the better speedup.

In Table 2 we present sample statistical data obtained from running the same benchmarks on a profiling version of Aurora with 8 workers on a Sequent Symmetry. The data gathered is: the number of procedure calls (including built-ins); the number of tasks (engine invocations); the number of choicepoints (nodes); the number of backtracks; the number of (conditional) bindings made by the engine; the number of nodes made public; the number of moves down (node by node) made by the scheduler; the number of bindings installed during moves down. For most items, we show the number of occurrences per task. Note that the number of procedure calls, nodes and backtracks is independent of the number of workers, but the other items (including bindings made by the engine) will vary.

The preliminary performance results that these tables illustrate are encouraging. On one processor, Aurora is only about 25% slower than Sicstus, the sequential system from which it is derived. Sicstus is itself only two times slower than Quintus Prolog, one of the fastest commercial systems. On 16 processors, the Aurora speedup (relative to its speed on one processor) is up to 14 on small programs with almost ideal or-parallelism, and substantial speedups are also obtained on larger pieces of code taken from real applications in natural language parsing and database query processing. It should be emphasised that these results are preliminary. The apparently slightly superlinear speedups on salt-must2 are probably due simply to variability in

[2] "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?"

the timing measurements.

These results demonstrate that the overhead introduced by adapting a high performance Prolog engine for the SRI model are low, and that significant speedups can be obtained on real examples. Both the timing data, and statistics on the number of bindings installed on task switching versus the number of ordinary bindings made, suggest that the overheads of updating binding arrays on task switching are quite tolerable in practice. Locking and other overheads associated with moving around the public part tree may be more of a problem, but this issue needs further investigation. Remarkably similar speedups on these same examples have been obtained for ECRC's PEPSys model [19]. The fact that two quite different models should produce similar speedups suggests that the speedups are limited mainly by the intrinsic parallelism in the examples.

As regards the ultimate goal of obtaining truly competitive bottom-line performance, Aurora is getting close but cannot yet be said to be an outright winner. We take as a point of comparison Quintus Prolog on a Sun 3/50. This is a fast commercial Prolog system, running on what is now only a moderately fast processor. On one Sequent Symmetry processor, Aurora is about 4 times slower, and on one processor of the (new) Encore Multimax, it is about 4.5 times slower. On 4 processors of either machine, Aurora performance nearly equals Quintus on most examples with sufficient parallelism. On the best examples with almost ideal or-parallelism, Aurora is about 2 to 3 times faster than Quintus, running on a 16-processor Encore. Of course, most examples don't have ideal parallelism, and it is possible to obtain higher sequential Prolog performance on machines faster than a Sun 3/50.

The main factor preventing Aurora from being truly competitive is that the infant technology of multiprocessor machines is finding it hard to keep pace with the dramatic yearly increase in sequential processor speeds. One processor on the latest Sequent or Encore machine is still 1.5 to 2 times slower than a Sun 3/50, which has now been surpassed by faster processors. The other factor is that the Aurora engine is about 2.75 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.

However, one can expect multiprocessors to become increasingly competitive both in absolute processor speed and price/performance. Then, with some tuning of the Aurora engine, it should certainly be possible to achieve truly competitive performance.

6 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, an existing, portable, state-of-the-art, 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.

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 processors 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 parallelization techniques to other existing Prolog systems.

Aurora supports the full Prolog language and is able to run large Prolog applications and systems. Examples include the Chat-80 natural language question answering system, and also the Sicstus Prolog compiler and environment (written in Prolog) which in fact forms part of Aurora system itself. The interesting question that is now open for exploration is to what extent real applications have enough exploitable parallelism *overall* for useful speedups to be obtainable[3].

7 FUTURE DIRECTIONS

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.

A proposed new data structure, the "wavefront" [3], promises to make scheduling more efficient and more flexible. The wavefront links all the active public nodes. The basic difference from other schemes is that all the dynamically changing scheduling information is to be found only in the wave-front, so that the rest of the public area becomes essentially read-only.

The current schedulers are able to handle cut, commit and side effects correctly. However, they require major enhancement to handle speculative work efficiently. The

[3]One should bear in mind the well-known Amdahl effect that the speedup can be no better than N if as little as 1 part in N of the computation is sequential.

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.

The existing Aurora system allows researchers to experiment with or-parallel logic programs. We intend to make the system available to other research groups. We expect to continue to add to its capabilities and speed, as the measurement tools needed for tuning are added, 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 [15, 27]. The work has also inspired ideas for novel architectures supporting shared virtual memory [25]. It is hoped that Aurora can contribute to the general 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, 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 A. M. Ali. Or-parallel execution of Prolog on BC-Machine. Sics research report, Swedish Institute of Computer Science, 1987.

[2] 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.

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

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

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

[6] Alan Calderwood. Aurora—the Manchester scheduler. Internal Report, Gigalips Project, January 1988.

[7] Mats Carlsson. Internals of Sicstus Prolog version 0.6. Internal Report, Gigalips Project, November 1987.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] Terrence Disz, Ewing Lusk, and Ross Overbeek. Experiments with OR-parallel logic programs. In Jean-Louis Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 576–600. MIT Press, 1987.

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

[15] Seif Haridi and David H. D. Warren. Andorra Prolog–the language and its application to distributed simulation. In *International Conference on Fifth Generation Computer Systems 1988.* ICOT, 1988.

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

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

[21] 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.

[22] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Rong Yang. Programming in Andorra-I. Internal Report, Gigalips Project, August 1988.