PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

OR-Parallel Speedups in a Knowledge Based System: on Muse and Aurora

Khayri A. M. Ali and Roland Karlsson

Swedish Institute of Computer Science, SICS
Box 1263, S-164 28 Kista, Sweden
khayri@sics.se and *roland@sics.se*

Abstract

The paper presents experimental results of running a knowledge based system that applies a set of rules to a circuit board (or a gate array) design and reports any design errors, on two OR-parallel Prolog systems, Muse and Aurora, implemented on a number of shared memory multiprocessor machines. The knowledge based system is written in SICStus Prolog, by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. When the system was tested on Muse and Aurora, without any modifications, the OR-parallel speedups were very encouraging as a large practical application. The number of processors used in our experiment is 25 on Sequent Symmetry (S81), 37 on BBN Butterfly II (TC2000), and 70 on BBN Butterfly I (GP1000). The results obtained show that the Aurora system is much more sensitive to the machine architecture than the Muse system, and the latter is faster than the former on all the three machines used. The real speedup factors of Muse, relative to SICStus, are 24.3 on S81, 31.8 on TC2000, and 46.35 on GP1000.

1 Introduction

Two main types of parallelism can be extracted from a Prolog program. The first, AND-parallelism, utilizes possibilities for simultaneous execution of several subproblems offered by Prolog semantics. The second, OR-parallelism, utilizes possibilities for simultaneous search for multiple solutions to a single problem. This paper is concerned with two systems exploiting only the latter type of parallelism: Muse [Ali and Karlsson 1990a] and Aurora [Lusk *et al.* 1990]. Both systems support the full Prolog language with its standard semantics, and they have been implemented on a number of shared multiprocessor machines, ranging from a few processors up to around 100 processors. Both systems show good speedups, in comparison with good sequential Prolog systems, for programs with a high degree of OR-parallelism. The two systems are based on two different memory models. Aurora is based on the SRI [Warren 1987] and Muse on incremental copying of the WAM stacks [Ali and Karlsson 1990a]. The two systems are implemented by adapting the same sequential Prolog system, SICStus version 0.6. The extra overhead associated with this adaptation is low and depends on the Prolog program and the machine architecture. For a large set of benchmarks, the average extra overhead for the Muse system on one processor is around 5% on Sequent Symmetry, 8% on BBN Butterfly GP1000, and 22% on BBN Butterfly TC2000. For the Aurora system with the same set of the benchmarks, it is around 25% on Sequent Symmetry, 30% on BBN Butterfly GP1000, and 77% on BBN Butterfly TC2000. Earlier results [Ali and Karlsson 1990b, Ali and Karlsson 1990c, Ali *et al.* 1991a, Ali *et al.* 1991b] show that the Muse system is faster than the Aurora system for a large set of benchmarks and on the above mentioned machines.

In this paper we investigate the performance results of Muse and Aurora systems on those multiprocessor machines for a large practical knowledge based system [Holmgren and Orsvärn 1989, Hagert *et al.* 1988]. The knowledge based system is used to check a circuit board (or a gate array) design with respect to a set of rules. These rules may for example be imposed by the development tool, by company standards or testability requirements. The knowledge based system has been written in SICStus Prolog [Carlsson and Widén 1988], by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. The gate array used in our experiment consists of 755 components. The system was tested on Muse and Aurora without any modifications. One important goal that has been achieved by Muse and Aurora systems is running Prolog programs that have OR-parallelism with almost no user annotations for getting parallel speedups.

The speedup results obtained are very good on all the machines used for the Muse system, but not for Aurora on the Butterfly machines. We found that this application has high OR-parallelism. In this paper we are going to present and discuss the results obtained from the Aurora and Muse systems on the three machines

used.

The paper is organized as follows. Section 2 briefly describes the three machines used in our experiment. Section 3 briefly describes the two OR-parallel Prolog systems, Muse and Aurora. Section 4 presents the knowledge based system. Sections 5 and 6 present and discuss the experimental results. Section 7 concludes the paper.

2 Multiprocessor Machines

The three machines used in our study are Sequent Symmetry S81, BBN Butterfly TC2000, and BBN Butterfly GP1000. Sequent Symmetry is a shared memory machine with a common bus capable of supporting up to 30 (i386) processors. Each processor has a 64-KByte cache memory. The bus supports cache coherence of shared data and its capacity is 80 MByte/sec. It presents the user with a uniform memory architecture and an equal access time to all memory.

The Butterfly GP1000 is a multiprocessor machine capable of supporting up to 128 processors. The GP1000 is made up of two subsystems, the processor nodes and the butterfly switch, which connects all nodes. A processor node consists of an MC68020 microprocessor, 4 MByte of memory and a Processor Node Controller (PNC) that manages all references. A non-local memory access across the switch takes about 5 times longer than local memory access (when there is no contention). The Butterfly switch is a multi-stage omega interconnection network. The switch on the GP1000 has a hardware supported block copy operation, which is used to implement the Muse incremental copying strategy. The peak bandwidth of the switch is 4 MBytes per second per switch path.

The Butterfly TC2000 is a similar to the GP1000 but is a newer machine capable of supporting up to 512 processors. The main differences are that the processors used in the TC2000 are the Motorola 88100s. They are an order of magnitude faster than the MC68020 and have two 16-KByte data and instruction caches. Thus in the TC2000 there is actually a three level memory hierarchy: cache memory, local memory and remote memory. Unfortunately no support is provided for cache coherence of shared data. Hence by default shared data are not cached on the TC2000. The peak bandwidth of the Butterfly switch on the TC2000 is 9.5 times faster than the Butterfly GP1000 (at 38 MBytes per second per path). The TC2000 switch does not have hardware support for block copy.

3 OR-Parallel Systems

In Muse and Aurora, OR-parallelism in a Prolog search tree is explored by a number of *workers* (processes or processors). A major problem introduced by OR-parallelism is that some variables may be simultaneously bound by workers exploring different branches of a Prolog search tree. Two different approaches have been used in Muse and Aurora systems for solving this problem. Muse uses incremental copying of the WAM stacks [Ali and Karlsson 1990a] while Aurora uses the SRI memory model [Warren 1987].

The idea of the SRI model is to extend the conventional WAM with a large binding array per worker and modify the trail to contain address-value pairs instead of just addresses. Each array is used by just one worker to store and access conditional bindings, i.e. bindings to variables which are potentially shareable. The WAM stacks are shared by all workers. The nodes of the search tree contain extra fields to enable workers to move around the tree. When a worker finishes a task, it moves over the tree to take another task. The worker starting a new task must partially reconstruct its array using the trail of the worker from which the task is taken.

The incremental copying of the WAM stacks used in Muse is based on having a number of sequential Prolog engines, each with its own local address space, and some global address space shared by all engines. Each sequential Prolog engine is a worker with its own WAM stacks. The stacks are not shared between workers. Thus, each worker has bindings associated with its current branch in its own copy of the stacks. This simple solution allows the existing sequential Prolog technology to be used without loss of efficiency. But it requires copying data (stacks) from one worker to another when a worker runs out of work. In Muse, workers incrementally copy parts of the (WAM) stacks and also share nodes with each other when a worker runs out of work. The two workers involved in copying will only copy the differing parts between the two workers states. The shared memory space stores information associated with the shared nodes on the search tree. Workers get work from shared nodes through using the normal backtracking mechanism of Prolog. Each worker having its own copy of the WAM stacks simplifies garbage collection, and caching the WAM stacks on machines, like the BBN Butterfly TC2000, that do not support cache coherence of shared data.

A node on a Prolog search tree corresponds to a Prolog choicepoint. Nodes are either *shared* or *nonshared (private)*. These nodes divide the search tree into two regions: *shared* and *private*. Each worker can be in either engine mode or in scheduler mode. In the engine mode, the worker works as a sequential Prolog system on private nodes, but is also able to respond to interrupt signals from other workers. Anytime a worker has to ac-

cess the shared region of the search tree, it switches to the scheduler mode and establishes the necessary coordination with other workers. The two main functions of a worker in the scheduler mode are to maintain the sequential semantics of Prolog and to match idle workers with the available work with minimal overhead.

The two systems, Muse and Aurora, have different working schedulers on the three machines used in our experiment. Aurora has two schedulers: the Argonne scheduler [Butler *et al.* 1988] and the Manchester scheduler [Calderwood and Szeredi 1989]. According to the reported results, the Manchester scheduler always gives better performance than the Argonne scheduler [Mudambi 1991, Szeredi 1989]. So, the Manchester scheduler will be used for Aurora in our experiment. Muse has only one scheduler [Ali and Karlsson 1990c, Ali and Karlsson 1991], so far.

The main difference between the Manchester scheduler for Aurora and the Muse scheduler is in the strategy used for dispatching work. The strategy used by the Manchester scheduler is that work is taken from the topmost node on a branch, and only one node at a time is shared. In Muse, several nodes at a time are shared and work is taken from the bottommost node on a branch. The bottommost strategy approximates the execution of sequential implementations of Prolog within a branch. Another difference between the two schedulers is in the algorithms used in the implementation of cut and side effects to maintain the standard Prolog semantics.

Many optimizations have been made of implementation of the Aurora and Muse systems on all the three machines. The only optimization that has been implemented for Muse and not for Aurora is caching the WAM stacks on the BBN Butterfly TC2000. In Aurora the WAM stacks are shared by the all workers while in Muse each worker has its own copy of the WAM stacks. Therefore, it is straightforward for Muse to make the WAM stack areas cachable whereas in Aurora it requires a complex cache coherence protocol to achieve this effect.

4 Knowledge Based System

One important process in the design of circuit boards and gate arrays is the checking of the design with respect to a set of rules. These rules may for example be imposed by the development tool, by company standards or by testability requirements. Until now, many of these rules have only been documented on paper. The check is performed manually by people who know the rules well. Increasing the number of gates in circuit boards (or in gate arrays) makes the manual check a very difficult process. Computerizing this process is very useful and may be the most reliable solution. The knowledge based systems group at SICS, in collaboration with groups from some Swedish companies, has been developing a knowledge based system that applies a set of rules to a circuit board (or a gate array) design and reports any design errors [Hagert *et al.* 1988, Holmgren and Orsvärn 1989]. The groups have developed two versions of the knowledge based system. The first version has been developed using a general purpose expert system shell while the second has been developed using SICStus Prolog. The latter, which will be used in our experiment, is more flexible and more efficient than the former. It is around 10 times faster than the first version on single processor machines. When it has been tested, without any modifications, on Muse and Aurora systems on Sequent Symmetry, the speedups obtained are linear up to 25 processors.

One reason for the high degree of OR-parallelism in this kind of application is that all of the rules applied to the circuit board (or a gate array) design are independent or could be made independent of each other. The second source of OR-parallelism is the application of each rule to all instances of a given circuit sub-assembly on the board. A circuit sub-assembly can be either a component (like *buffer, inverter, nand, and, nor, or, xor,* etc.) or a group of interconnected components. The knowledge based system mainly consists of an inference engine, design rules, and a database describing the circuit board (or the gate array). The inference engine is implemented as a metainterpreter with only 8 Prolog clauses. The gate array used in our experiment consists of 755 components (Texas gate array family TGC–100), which is described by around 10000 Prolog clauses. The design rules part with its interface to the gate array description is around 200 Prolog clauses. Eleven independent rules are used in this experiment. The metainterpreter applies the set of rules to the gate array description. For a larger gate array more OR-parallelism is expected. It should be mentioned that people who developed the knowledge based system did not at all consider parallelism, but they tried to make their system easy to maintain by writing clean code. They avoided using side effects, but they have used cuts (embedded in If_Then_Else) and findall constructs. The user interface part of this application is not included in our experiment.

Since Muse and the Aurora system are also running on larger machines, the BBN Butterfly machines, it was more natural to test the knowledge based system on those machines. The speedup results obtained differ for the Muse and the Aurora system. On 37 TC2000 processors, Muse is 31.8 times faster than SICStus, while Aurora is only 7.3 times faster than SICStus. Similarly, on 70 GP1000 processors Muse is 46.35 times faster than SICStus, while Aurora is only 6.68 times faster than SICStus. The low speedup for the Aurora system is surprising since this application is rich in OR-parallelism. Is this a scheduler problem for Aurora or an engine problem? The following two sections are going to present and

analyze the results of Muse and Aurora, in order to try to answer this question.

5 Timings and Speedups

In this section we present timing and speedup results obtained from running the knowledge based system on Muse and Aurora systems. The runtimes given in this paper are the **mean values obtained from eight runs**. On Sequent Symmetry, there is no significant difference between mean and best values, whereas on the Butterfly machines, mean values are more reliable than best values due to variations of timing results from one run to another[1]. Variations around the mean value will be shown in the graphs by a vertical line with two short horizontal lines at each end. The speedups given in this section are relative to running times of Muse on one processor on the corresponding machine. The SICStus one-processor runtime on each machine will also be presented to determine the extra overhead associated with adapting the SICStus Prolog system to the Aurora and Muse systems. Sections 5.1, 5.2, and 5.3 present those results on Sequent Symmetry, GP1000, and TC2000 machines, respectively.

5.1 Sequent Symmetry

Table 1 shows the runtimes of Aurora and Muse on Sequent Symmetry, and the ratio between them. Times are shown for 1, 5, 10, 15, 20, and 25 workers with speedups (relative to one Muse worker) given in parentheses. The SICStus runtime on one Sequent Symmetry processor is 422.39 seconds. This means that for this application and on the Sequent Symmetry machine the extra overhead associated with adapting the SICStus Prolog system to Aurora is 26.3%, and for Muse is only 1.0% (calculated from Table 1). The performance results that Table 1 illustrates are good for both systems, and Aurora timings exceed Muse timings by 25% to 26% between 1 to 25 workers. Figure 1 shows speedup curves for Muse and Aurora on Sequent Symmetry. Both systems show linear speedups with no significant variations around the mean values.

Table 1: Runtimes (in seconds) of Aurora and Muse on Symmetry, and the ratio between them.

Workers	Aurora	Muse	Aurora/Muse
1	533.69(0.80)	426.74(1.00)	1.25
5	106.87(3.99)	85.67(4.98)	1.25
10	53.58(7.96)	42.94(9.94)	1.25
15	36.06(11.8)	28.73(14.9)	1.26
20	27.22(15.7)	21.65(19.7)	1.26
25	21.83(19.5)	17.39(24.5)	1.26

[1]These variations are due mainly to switch contention.

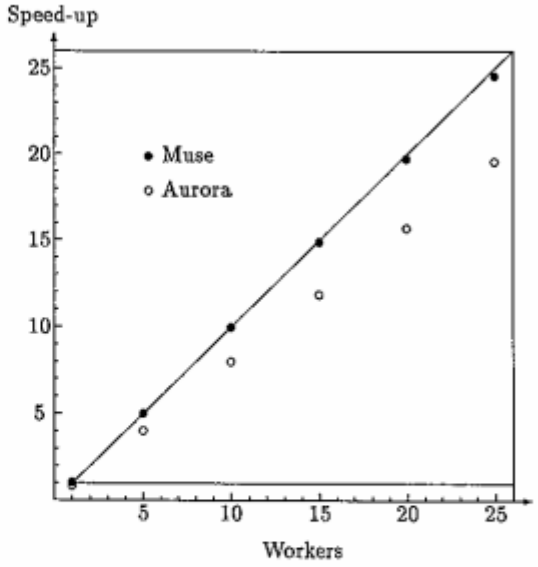

Figure 1: Speedups of Muse and Aurora on Symmetry, relative to 1 Muse worker.

5.2 BBN Butterfly GP1000

Table 2 shows the runtimes of Aurora and Muse on GP1000 for 1, 10, 20, 30, 40, 50, 60, and 70 workers. The SICStus runtime on one GP1000 node is 534.4 seconds. So, for this application and on the GP1000 machine the extra overhead associated with adapting the SICStus Prolog system to Aurora is 66%, and for Muse is only 7%. Here the performance results are good for the Muse system but not for the Aurora system. Aurora timings are longer than Muse timings by 55% to 594% between 1 to 70 workers.

Figure 2 shows speedup curves corresponding to Table 2 with variations around the mean values. The speedup curve for Aurora levels off beyond around 20 workers. On the other hand, the Muse speedup curve levels up as more workers are added.

Table 2: Runtimes (in seconds) of Aurora and Muse on GP1000, and the ratio between them.

Workers	Aurora	Muse	Aurora/Muse
1	886.4(0.65)	572.3(1.00)	1.55
10	105.3(5.44)	58.3(9.82)	1.81
20	74.1(7.72)	29.8(19.2)	2.49
30	72.7(7.88)	20.7(27.7)	3.52
40	64.3(8.91)	16.1(35.5)	3.99
50	72.4(7.90)	13.8(41.6)	5.26
60	65.7(8.71)	12.4(46.1)	5.29
70	80.0(7.15)	11.5(49.6)	6.94

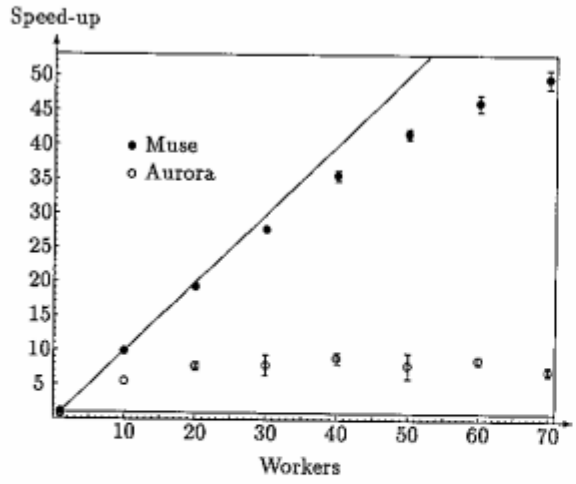

Figure 2: Speedups of Muse and Aurora on GP1000, relative to 1 Muse worker.

5.3 BBN Butterfly TC2000

Table 3 shows the performance results of Aurora and Muse on TC2000 for 1, 10, 20, 30, and 37 workers. The SICStus runtime on one TC2000 node is 100.48 seconds. Thus, for this application and on the TC2000 machine the extra overhead associated with adapting the SICStus Prolog system to Aurora is 80%, and for Muse is only 5%. Here also the performance results are good for the Muse system but not for the Aurora system. Aurora timings are longer than Muse timings by 70% to 319% between 1 to 37 workers.

Table 3: Runtimes (in seconds) of Aurora and Muse on TC2000, and the ratio between them.

Workers	Aurora	Muse	Aurora/Muse
1	180.55(0.59)	105.97(1.00)	1.70
10	22.12(4.79)	10.81(9.80)	2.05
20	16.02(6.61)	5.56(19.1)	2.88
30	13.66(7.76)	3.93(27.0)	3.48
37	13.79(7.68)	3.29(32.2)	4.19

Figure 3 shows speedup curves corresponding to Table 3. The speedup curves are similar to the corresponding ones shown in Figure 2.

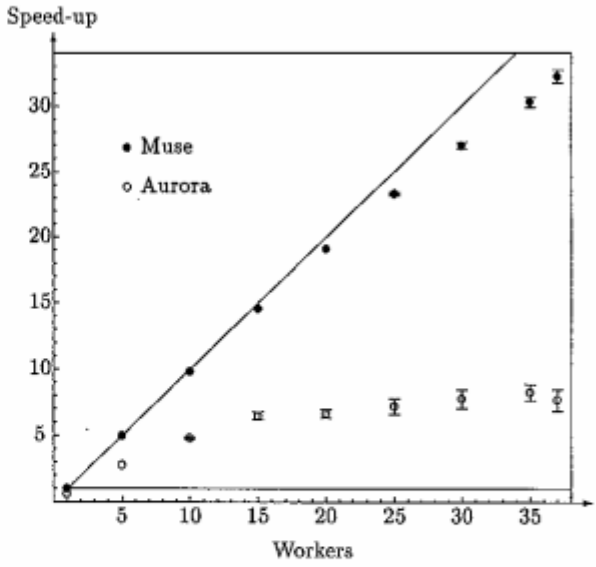

Figure 3: Speedups of Muse and Aurora on TC2000, relative to 1 Muse worker.

6 Analysis of Results

From the results presented in Section 5 we found that the Muse system shows good performance results on the three machines, whereas the Aurora system shows good results only on the Sequent Symmetry. In this section, we try to explain the reason for these results by studying the Muse and Aurora implementations on one of the Butterfly machines (TC2000). The TC2000 has better support for reading the realtime clock than the GP1000. A worker time could be divided into the following three main activities:

1. *Prolog:* time spent executing Prolog (i.e., engine time).
2. *Idle:* time spent waiting for work to be generated when there is temporarily no work available in the system.
3. *Others:* time spent in all the other activities (i.e., all scheduling activities) like spin lock, signalling other workers, performing cut, grabbing work, sharing work, looking for work, binding installation (and copying in Muse), synchronization between workers, etc.

Table 4 and Table 5 show time spent in each activity and the corresponding percentage of the total time. Results shown in Table 4 and Table 5 have been obtained from instrumented versions of Muse and Aurora on the TC2000. The times obtained from the instrumented versions are longer than those obtained from

Table 4: Time (in seconds) spent in the main activities of Muse workers on TC2000.

Muse Workers	Activity		
	Prolog	Idle	Others
1	128.36(100)	0	0
5	128.80(99.7)	0.09(0.1)	0.26(0.2)
10	129.28(99.1)	0.40(0.3)	0.71(0.5)
20	129.90(96.5)	3.56(2.6)	1.17(0.9)
30	130.32(95.4)	4.17(3.0)	2.11(1.5)

Table 5: Time (in seconds) spent in the main activities of Aurora workers on TC2000.

Aurora Workers	Activity		
	Prolog	Idle	Others
1	210.42(98.2)	0	2.36(1.1)
5	221.24(98.3)	0.19(0.1)	2.03(0.9)
10	235.34(98.1)	0.43(0.2)	2.43(1.0)
20	329.60(98.1)	1.11(0.3)	3.61(1.1)
30	412.97(94.7)	13.70(3.2)	7.64(1.8)

uninstrumented systems by around 19–27%. So, they might not be entirely accurate, but they help in indicating where most of the overhead is accrued.

Before analyzing the data in Table 4 and Table 5 we would like to make two remarks on these data. The first remark is that in the Aurora system the overhead of checking for the arrival of requests is separated from the Prolog engine time, while in the Muse system there is no such separation. This explains why there is scheduling overhead (*Others*) in the 1 worker case in Table 5 and not in Table 4. The other remark is that the figures obtained from the Aurora system do not total 100% of time, since a small fraction of the time is not allocated to any of the three activities. However, these two factors have no significant impact on the following discussion.

By careful investigation of Table 4 and Table 5 we find that the total *Prolog* time of Muse workers is almost constant with respect to the number of workers whereas the corresponding time for Aurora grows rapidly as new workers are added. We also find that the scheduling time (*Others*) in Table 5 is not very high in comparison with the corresponding time in Table 4. Similarly, the difference of *Idle* time between Muse and Aurora is not so high. So, the main reason for performance degradation in Aurora is the Prolog engine speed.

We think that the only factor that slows down the Aurora engine as more workers are added is the high access cost of non-local memory. Non-local memory access takes longer time than local memory access, and causes switch contention. Non-local memory accesses can be due to either the global Prolog tables or the WAM stacks. In Muse and Aurora systems, the global tables are partitioned into parts and each part resides in the local memory of one processor. In Aurora the WAM stacks are shared by all workers while in Muse each worker has its own copy of the WAM stacks. The global Prolog tables have been implemented similarly in the both Muse and Aurora systems. Since the Muse engine does not have any problem with the Prolog tables, the problem should lie in the sharing of the WAM stacks in Aurora, coupled with the fact that this application generates around 9.8 million conditional bindings, and executes around 1.1 million Prolog procedure calls. On the average, each procedure call generates around 9 conditional bindings. This may mean that the reason why Aurora slows down lies in the cactus stack approach, which causes a great many non-local accesses to the Prolog stacks. This results in a high amount of switch contention once over five workers. This is avoided in the Muse model, since each worker has its own copy of the WAM stacks in the processor local memory and the copy is even cachable. Unfortunately, we could not verify this hypothesis because the current Aurora implementation on the TC2000 does not provide any support for measuring the stack variables access time.

7 Conclusions

Experimental results of running a large practical knowledge based system on two OR-parallel Prolog systems, Muse and Aurora, have been presented and discussed. The number of processors used in our experiment is 25 on Sequent Symmetry (S81), 37 on BBN Butterfly II (TC2000), and 70 on BBN Butterfly I (GP1000). The knowledge based system used in our study checks a circuit board (or a gate array) design with respect to a set of rules and reports any design errors. It is written in SICStus Prolog, by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. It is used in our experiment without any modifications.

The results of our experiment show that this class of applications is rich in OR-parallelism. Very good real speedups, in comparison with SICStus Prolog system, have been obtained for the Muse system on all three machines. The real speedup factors for Muse are 24.3 on 25 S81 processors, 31.8 on 37 TC2000 processors, and 46.35 on 70 GP1000 processors. The obtained real speedup factors for Aurora are lower (than for Muse) on Sequent Symmetry, and much lower on the Butterfly machines. The Aurora timings are longer than Muse timings by 25% to 26% between 1 to 25 S81 processors, 70% to 319% between 1 to 37 TC2000 processors, and 55% to 594% between 1 to 70 GP1000 processors.

The analysis of the obtained results indicates that the main reason for this great difference between Muse timing and Aurora timing (on the Butterfly machines) lies in the Prolog engine and not in the scheduler. The Aurora engine is based on the SRI memory model in

which the WAM stacks are shared by the all workers. We think that the only reason why the Aurora engine slows down as more workers are added is to be found in the large number of non-local accesses of stack variables. This results in a high amounts of switch contention as more workers are added. This is avoided in the Muse model, since each worker has its own copy of the WAM stacks in the processor local memory and even cachable in the TC2000. Unfortunately, we could not verify this hypothesis because the current Aurora implementation on the Butterfly machines does not provide any support for measuring access time of stack variables.

8 Acknowledgments

We would like to thank the Argonne National Laboratory group for allowing us to use their Sequent Symmetry and Butterfly machines. We thank Shyam Mudambi for his work on porting Muse and Aurora to the Butterfly machines. We also would like to thank Fredrik Holmgren, Klas Orsvärn and Ingvar Olsson for discussions and allowing us to use their knowledge based system.

References

[Ali and Karlsson 1990a] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, pages 129–162, Vol. 19, No. 2, April 1990.

[Ali and Karlsson 1990b] Khayri A. M. Ali and Roland Karlsson. The Muse OR-Parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776, MIT Press, October 1990.

[Ali and Karlsson 1990c] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *International Journal of Parallel Programming*, pages 445–475, Vol. 19, No. 6, Dec. 1990.

[Ali and Karlsson 1991] Khayri A. M. Ali and Roland Karlsson. Scheduling OR-Parallelism in Muse. In *Proceedings of the 1991 International Conference on Logic Programming*, pages 807–821, Paris, June 1991.

[Ali *et al.* 1991a] Khayri A. M. Ali, Roland Karlsson and Shyam Mudambi. Performance of Muse on the BBN Butterfly TC2000. In *Proceedings of the ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, June 1991. To appear also in *Lecture Notes in Computer Science*, Springer Verlag.

[Ali *et al.* 1991b] Khayri A. M. Ali, Roland Karlsson and Shyam Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. Submitted to the *New Generation Computing Journal*, 1991.

[Butler *et al.* 1988] 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 and Symposium on Logic Programming*, pages 1590–1605, MIT Press, August 1988.

[Calderwood and Szeredi 1989] 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.

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

[Hagert *et al.* 1988] G. Hagert, F. Holmgren, M. Iidell and K. Orsvärn. On Methods for Developing Knowledge Systems—an Example in Electronics, Mekanresultat 88003 (in Swedish), Sveriges Mekanförbund, Box 5506, 114 85 Stockholm, 1988.

[Holmgren and Orsvärn 1989] Fredrik Holmgren and Klas Orsvärn. Towards a Domain Specific Shell for Design Rule Checking. In *Proceedings of the IFIP TC 10/WG10.2 Working Conference on the CAD Systems Using AI Techniques*. pages 221–228, Tokyo, June 6–7, 1989.

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

[Mudambi 1991] Shyam Mudambi. Performance of Aurora on NUMA machines. In *Proceedings of the 1991 International Conference on Logic Programming*, pages 793–806, Paris, June 1991.

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

[Warren 1987] 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.