Parallel Conventional Systems versus Parallel Logic Programming Systems on Distributed Shared Memory Architectures *

Vanusa Menditi Calegario
Inês de Castro Dutra

Department of Systems Engineering and Computer Science
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
e-mail: {vanusa,ines}@cos.ufrj.br

Abstract

Distributed shared memory architectures have been object of research by many computer science groups. Research goes broadly from hardware based coherence protocols to DSM software protocols on networks of workstations passing through high technology interconnection networks that reduce network latency. In this work we thoroughly investigate how different hardware cache coherence protocols affect performance of parallel logic programming systems and compare to results obtained for parallel conventional systems.

We use execution-driven simulation of a hardware DSM (DASH) to investigate the access patterns and caching behaviour exhibited by parallel C programs and by Aurora, a parallel logic programming system capable of exploiting implicit parallelism in Prolog programs. Aurora was written originally to run on bus-based shared memory platforms.

The simulator allows us to vary several machine settings such as cache blocksize, cache size, network path width, write buffer depth as well as allows us to test different cache coherence protocols. Our work concentrates on the study of Prolog and parallel C programs under different cache coherence protocols regarding basic performance, scalability, and programmability.

Our results indicate that cache coherence protocols do not significantly affect regular applications while irregular ones can benefit from hybrid or invalidate protocols. Our results also show that parallel logic programming systems written for shared memory machines can achieve good scalability on modern architectures, but at the cost of high time and space complexity when compared to conventional C programs. Results also indicate that optimisations such as improvements on the abstract engine and scheduling can make parallel logic programming systems competitive and more attractive than parallel C systems for some applications, specially if we consider that parallel logic programming systems exploit implicit parallelism.

Keywords: logic programming, conventional programming, parallelism, DSM architectures, performance evaluation

1 Introduction

Distributed shared memory architectures have been object of research by many computer science groups. Research goes broadly from hardware based coherence protocols to DSM software protocols on networks of workstations passing through high technology interconnection networks that reduce network latency. In this work we thoroughly investigate how different hardware cache coherence protocols affect performance of parallel logic programming systems and compare to results obtained for parallel conventional systems.

We use execution-driven simulation of a hardware DSM (DASH [16]) to investigate the access patterns and caching behaviour exhibited by parallel C programs and by Aurora [17], a parallel logic programming system capable of exploiting implicit parallelism in Prolog programs. Aurora was written originally to run on bus-based shared memory platforms.

The simulator allows us to vary several machine settings such as cache blocksize, cache size, network path width, write buffer depth as well as allows us to test different cache coherence protocols. Our work concentrates on the study of Prolog and parallel C programs under different cache coherence protocols regarding basic performance, scalability, and programmability.

Our results indicate that cache coherence protocols do not significantly affect regular applications while irregular ones can benefit from hybrid or invalidate protocols. Our results also show that parallel logic programming systems written for shared memory machines can achieve good scalability on modern

*Research supported by CNPq and Capes, Brazilian Research Councils

architectures, but at the cost of high time and space complexity when compared to conventional C programs. However, results also indicate that optimisations such as improvements on the abstract engine and scheduling can make parallel logic programming systems competitive and more attractive than parallel C systems for some applications, specially if we consider that parallel logic programming systems exploit implicit parallelism.

Our work contrasts with previous studies of the performance of coherence protocols for parallel logic programming systems. Tick and Hermenegildo [29] studied caching behaviour of independent and-parallelism in bus-based multiprocessors. Other researchers have studied the performance of parallel logic programming systems on scalable architectures, such as the DDM [20], but did not evaluate the impact of different coherence protocols. Also they did not tackle the differences between parallel logic programming systems and parallel C systems.

Recent work has tackled the problem of whether logic programming systems can obtain good performance on scalable architectures with a particular machine setting [22, 21, 24], but without any comparison with parallel C systems. Silva et al tackled the problem of how architectural parameters can affect the performance of parallel logic programming systems [25].

Another similar work we can mention is the Aquarius Prolog Compiler, that compares sequential Prolog with sequential C performance. A comparison with C was done using four programs: tak, fibonacci, hanoi and quicksort [30]. Other related work on comparison of sequential C and Prolog systems was done on the evaluation of different Prolog implementation techniques [6].

Regarding programmability and performance comparison between systems that exploit implicit and explicit parallelism, a related work evaluates parallelism on shared memory architectures using Sisal, a functional language with implicit parallelism, SR, an imperative language with explicit parallelism and C. Five **scientific** applications were programmed in each language and evaluated for performance and programmability. The author basically concludes that implicit parallelism is better if the compiler creates an efficient program, whereas, explicit parallelism is better when there is a considerable improvement on performance. When the performance is similar, the ease of programming favours implicit parallelism [9].

Our work differs from this in that we evaluate our benchmark set on distributed shared memory architectures and evaluate also symbolic applications instead of only scientific ones.

The paper is organised as follows. Section 2 introduces concepts of parallel logic programming systems and briefly discusses implicit and explicit parallelism. In section 3 we describe our work methodology. Section 4 describes the applications and algorithms used and results obtained. Finally, section 5 concludes this work and draws our next steps.

2 Implicit parallelism x Explicit Parallelism

Two main sources of implicit parallelism can be exploited in logic programming: (1) and-parallelism (at goal level) and (2) or-parallelism (at clause level). Parallel logic programming systems can exploit one or both forms of parallelism [17, 35, 11].

Each kind of parallelism has its own advantages and disadvantages, but we have chosen to focus on OR-parallelism as a first step for a number of reasons. From our perspective, the wide range of potential applications and the very slight execution overhead have been the main reasons. See [17] for a fuller discussion about the advantages of OR-parallelism in the context of the Aurora system.

Because of its declarative nature, a Prolog-like program can be parallelised without any change to the Prolog source code. Logic programming runtime systems are responsible for managing and controlling the parallelism. This approach has significant advantages when compared to explicit parallelism because the Prolog programmer does not need to worry about issues as synchronisation, communication and scheduling. Therefore the exploitation of implicit parallelism allows portability.

Some applications have very irregular computational patterns that make them hard to run efficiently on parallel machines. If they are written in C, aspects such as dynamic load balancing need to be taken into account while in parallel logic programming systems that exploit implicit parallelism this is accomplished by the runtime system.

3 Methodology

In this section we detail the methodology used in our experiments. The experiments consisted of the simulation of the parallel execution of Aurora and C programs, compiled for the MIPS architecture.

3.1 Aurora

Aurora is an or-parallel implementation of the full Prolog language for shared memory multiprocessors that uses the SRI model [34] to represent different bindings of the same logical variable corresponding to different branches of the search space. In the SRI model, a group of workers (processing elements) 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. 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 real work, and corresponds 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. Workers try to maximise the time they spend working and minimise the time they spend scheduling. In order to share or-parallel work, Aurora protects the nodes (choicepoints) with locks avoiding that several workers steal the same piece of work.

The scheduler used in our experiments is the Bristol scheduler. Several strategies have been used by the Bristol or-scheduler, but our results use the version that employs the leftmost and richest worker selection [3]. In this strategy **busy** workers every so often voluntarily suspend seeking for less speculative work in the leftmost part of the tree. Speculative work corresponds to work that may be pruned later because of the execution of a Prolog pruning operator by a processor in another branch. **Idle** workers seek for work from the richest workers and take work from the bottom most part of the tree to maintain locality. It uses pruning operators counters and scope [12] to figure out which work is speculative and which is not [4]. The main algorithm of the Bristol scheduler follows the basic algorithm mentioned before, where workers always give preference to mandatory work, and are always seeking for better work if they are working in any speculative work.

3.2 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 32-node, DASH-like [16], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a write buffer, a direct-mapped data cache, local memory, a full-map directory, and a network interface. The simulator was developed at the University of Rochester and uses the MINT front-end, developed by Veenstra and Fowler [31], to simulate the MIPS architecture, and a back-end, developed by Bianchini and Veenstra [5], to simulate the memory and inter-connection systems.

In our simulated machine, each processor has a 64-KB direct-mapped data cache with 64-byte cache blocks. All instructions and read hits are assumed to take 1 cycle. Read misses stall the processor until the read request is satisfied. Writes go into a write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. Shared data are interleaved across the memories at the block level.

A memory bus clocked at half of the speed of the processor connects the main components of each machine node. A new bus operation can start every 34 processor cycles. A memory module can provide the first word of a cache line 20 processor cycles after the request is issued. The other words are delivered at 2 cycles/word bandwidth.

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor clock speed. Switch nodes introduce a 4-cycle delay to the header of each message. Network paths are 16-bit wide, which matches the memory bandwidth. In these networks contention for links and buffers is captured at the source and destination of messages.

All hardware characteristics mentioned above are common in actual modern parallel architectures.

In order to keep caches coherent we used write-invalidate (WI) [10], write-update (WU) [19] and dynamic hybrid [14] protocols. In the WI protocol, whenever a processor writes a data item, copies of the cache block containing the item in other processors' caches are invalidated. If one of the invalidated processors later requires the same item, it will have to fetch it from the writer's cache. Our WI protocol keeps caches coherent using the DASH protocol with release consistency [15].

WU protocols are the main alternative to invalidate-based protocols. In WU protocols, whenever an item is written, the writer sends copies of the new value to the other processors that share the item. In our WU implementation, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowl-

edgement to the writing processor. The writing processor only stalls waiting for acknowledgements at a lock release point.

Our WU protocol implementation includes two optimisations. First, when the home node receives an update for a block that is only cached by the updating processor, the acknowledgement of the update instructs the processor to retain future updates since the data is effectively private. Second, when a parallel process is created by *fork*, we flush the cache of the parent's processor, which eliminates useless updates of data initialised by the parent but not subsequently needed by it.

In order to reduce the number of update messages of the WU protocol, we experimented with a dynamic hybrid protocol [14] based on the coherence protocols of the bus-based multiprocessors using the DEC Alpha AXP21064 [28]. In these multiprocessors, each node makes a local decision to invalidate or update a cache block when it sees an update transaction on the bus. We associate a counter with each cache block and invalidate the block when the counter reaches the threshold. References to a cache block reset the counter to zero. We used counters with thresholds of 1 (Hyb1) and 2 (Hyb2) updates.

In our simulated results, for each application, parallel performance is directly related to cache read miss rates and network traffic caused by each change in number of processors and cache coherence protocols.

For WI, within a column, misses are classified as:

- **Cold start misses.** A cold start miss happens on the first reference to a block by a processor.
- **True sharing misses.** A true sharing miss happens when a processor references a word belonging in a block it had previously cached but has been invalidated, due to a write by some other processor to the same word.
- **False sharing misses.** A false sharing miss occurs in roughly the same circumstances as a true sharing miss, except that the word written by the other processor is not the same as the word missed on.
- **Eviction misses.** An eviction (replacement) miss happens when a processor replaces one of its cache blocks with another one mapping to the same cache line and later needs to reload the block replaced.

This classification uses the algorithm described in [8], as extended in [5]. Note that the WU protocol does not have sharing misses.

The hybrid protocol includes an extra class for the miss categorisation, `drop` misses, to account for cache misses resulting from excessively eager self-invalidations.

We classify updates as either *useful* (also known as **true sharing updates**), which are needed for correct execution of the program, or *useless*. The former category is used when the receiving processor references the word modified by the update message before another update message to the same word is received. The latter category of updates includes:

- **False sharing updates.** The receiving processor does not reference the word modified by the update message before it is overwritten by a subsequent update, but references some other word in the same cache block.
- **Proliferation updates.** The receiving processor does not reference the word modified by the update message before it is overwritten, and it does not reference any other word in that cache block either.
- **Replacement updates.** The receiving processor does not reference the updated word until the block is replaced in its cache.
- **Termination updates.** A termination update is a proliferation update happening at the end of the program.

This classification uses the algorithm described in [5]. The categorisation is fairly straightforward, except for the false update class. Successive (useless) updates to the same word in a block are classified as proliferation instead of false sharing updates, if the receiving processor is not concurrently accessing other words in the block. Thus, the algorithm classifies useless updates as proliferation updates, unless *active* false sharing is detected or the application terminates execution.

The categorisation of updates for the hybrid protocol includes an additional category, `drop` updates, to account for the updates that cause blocks to be invalidated.

3.3 The Parallel Environment

In order to use Aurora with the simulator we needed to port the system to the MIPS architecture. We used the FSF's `gcc 2.7.2` C compiler and `binutils-2.6` assembler and linker under a Solaris 2.4 environment as cross development tools for this purpose.

We also cross-compiled our parallel C programs for the MIPS architecture using the same tools.

We use the IRIX 4.0 support for shared memory and synchronisation implemented at the simulator level. The simulator has an optimisation when implementing spinlocks. When a process tries to grab a lock, if it fails, the simulator blocks the process till the lock is released by another processor [32].

The Aurora and the C programs were compiled with the `-O2` option.

4 Applications and Results

We concentrated our studies on some applications commonly used as benchmarks for Prolog or C systems. Due to lack of space we evaluated only four applications. These four applications are simple, easy to understand and to write the same algorithm in C and Prolog. Moreover, the Prolog applications were chosen to have or-parallelism, because Aurora only exploits or-parallelism. A complete benchmark set can be found elsewhere [6]. We tried to write the same sequential algorithm in each language in order to obtain a more accurate comparison. Both C and Prolog algorithms always give the same answer. It is important to notice that all C programs needed to be parallelised in order to run in several processors while the Prolog applications were not modified. The Prolog and C source codes for the programs can be found at `http://www.cos.ufrj.br/~vanusa/benchs.html`.

4.1 Applications and Algorithms

Matrix Multiplication This is a typical scientific application that computes $C = A \times B$, where A, B and C are matrices and A and B are two input matrices that have the same shape and contents. The C program was taken from the Splash benchmark [26]. We ran the programs for a 150x150 matrix size. The matrices are represented in Prolog through database facts. The C parallel version of this program considers matrix A to be private and matrices B and C to be shared. The algorithm tries to evenly allocate chunks of the matrix A among the processors, in order that each computes the same amount of work when possible.

N-Queens Problem The N-queens problem is a classical combinatorial problem in the artificial intelligence area. The problem is to place N queens on an N x N chessboard so that no two queens attack each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal. This problem is commonly used as a benchmark for algorithms that solve constraint satisfaction problems (CSP) [18]. It has found many useful, practical scientific and engineering applications including 2-dimensional VLSI routing and testing, maximum full range communication, traffic control and parallel optical computing [27].

The algorithms used in Prolog and in C generates all solutions. Both programs implement the same algorithm that tries to place each queen on the board at a time checking if the current placement threatens the previous ones. The Prolog program was taken from the Aurora benchmark. The C program was written by ourselves. We ran the programs for a 10x10 board size.

This problem is well known to generate exponential growth of the search space. Usually, such problems can run with larger input data when parallelised. However this problem is very difficult to parallelise using a procedural approach. Our C program was parallelised by attaching to each processor one top square of the board. Thus limiting the maximum number of processors to have some work to do by N (board size). For example, in a 4x4 board, using 4 processors, processor 0 would compute the solution with a queen fixed at position (0,0), processor 1 would compute the solution with a queen fixed at position (1,0), processor 2 would compute the solution with a queen fixed at position (2,0), and processor 3 would compute the solution with a queen fixed at position (3,0). Obviously, this solution produces a lot of load imbalance.

Once more, our Prolog program was not modified to run in parallel.

A Database Example This is a simple database program that represents family and employment relations. A group of 32 queries was used as input. The C program searches the database sequentially

and iteratively. Both programs were written by graduate students and adapted to our purposes. The database has two thousand records.

We parallelised the C program in a simple way by allocating each query to each processor.

Traveling Salesperson Problem The traveling salesperson problem is a very popular optimisation problem that consists of visiting all nodes in a graph at most once in order to find a path with minimum cost. The C program represents the partial tours as a fibonacci heap data structure to improve performance. It uses a branch and bound method based on the A* search algorithm. The Prolog program uses the same algorithm, but represents the graph as a set of facts. Besides this, the Prolog program generates the paths through intelligent permutation of the nodes in the graph. The C program was taken from the Splash benchmark [26]. The Prolog version was developed by ourselves. We used a 8-node graph as input (a 16-node graph runs for days on a Ultra1 sparc machine on the simulator!).

4.2 Simulated Results

4.2.1 Sequential Times

Application	Invalidate	Update	Hybrid 1	Hybrid 2	P/C
Matrix 150 P	1365311633	1365311633	1365311633	1365311633	
Matrix 150 C	60522671	60522671	60522671	60522671	22.56
Queens 10 P	921882260	921882260	921882260	921882260	
Queens 10 C	21829989	21829989	21829989	21829989	42.23
TSP 8 P	6327860	6327860	6327860	6327860	
TSP 8 C	444210	444210	444210	444210	14.25
Database P	264341308	264341308	264341308	264341308	
Database C	5627418	5627418	5627418	5627418	46.97

Table 1: Simulated Results - **Aurora** and **C** Elapsed Simulated Cycles - 1 processor

Table 1 shows the sequential execution times for the C and Aurora Prolog versions of the programs for the four protocols. Times are given in number of simulated cycles. C programs are followed by the 'C' letter and Prolog programs are followed by the 'P' letter. The column P/C shows the ratio between Aurora and C execution times. We can observe that the C programs run from 14 to 47 times faster than Aurora.

Aurora is around 7 times slower than one of the best sequential Prolog implementations known, SICStus 3.0 [7] (Aurora is based on SICStus 0.6). It also does not generate native code, which slows it down by a factor of three [6]. Therefore if we consider Aurora using a faster engine, we could predict an improvement of around 20 times in speed.

As expected, in a one processor simulation, the execution times are not affected by different protocols.

These sequential results show that Aurora needs several optimisations in order to be competitive with conventional C systems. The issues of optimisation will be discussed later. We will see in the next section that the inefficiency of the system is counterbalanced by the implicit parallelism exploited from sequential Prolog programs.

4.2.2 Scalability

In this section we show results for Aurora and parallel C running the four applications with 2, 4, 8, 16, 24 and 32 processors using four different cache coherence protocols.

Figures 1, 5, 9, 11 show speedup graphs respectively for the matrix, queens, tsp and database applications. Notice that base times to compute speedups are different for C and Aurora. Graphs were built in order to show how Aurora and C individually manage to efficiently exploit all parallelism available in the applications.

Matrix Multiplication The matrix multiplication program written in Prolog has or-parallelism that stems from choicepoints created for the calculation of each element of matrix C. An outline of the search

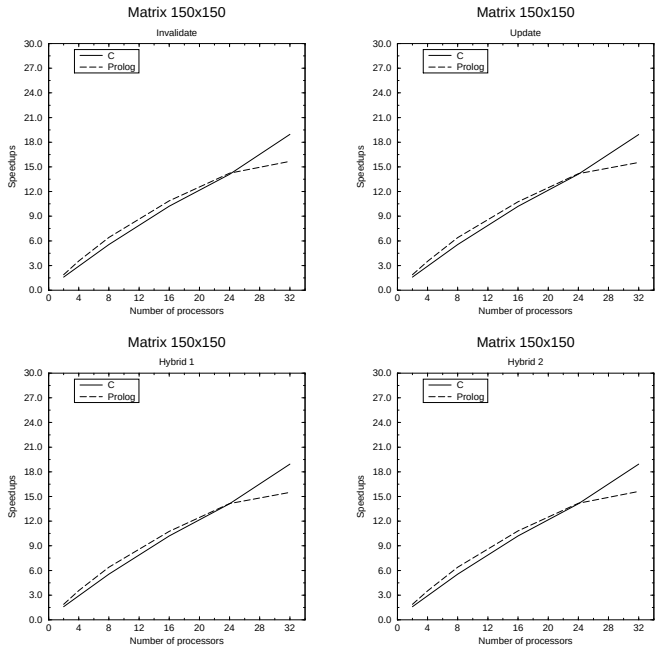

Figure 1: Matrix speedups, four protocols

space is shown in figure 2. A choicepoint with 150 alternatives is created at the root node of the tree and on each of these 150 branches (each line of the matrix C) a choicepoint of size 150 is created. This parallelism is implicitly and efficiently exploited by Aurora which makes the system achieve slightly better speedups than parallel C up to 24 processors as can be observed in figure 1.

As we increase the number of processors the system does not scale up. When we observe the execution in more detail, we find out that some processors in the 32-processor simulation stay blocked much longer than the 24-processor simulation. This indicates that there is contention to access a lock with processors trying to grab work always from the same worker. This also indicates that the or-scheduler version we use for Aurora is not scalable due to the way workers try to fetch work from other workers. In contrast, the C parallel program scales well up due to the static load distribution of the matrices among the processors.

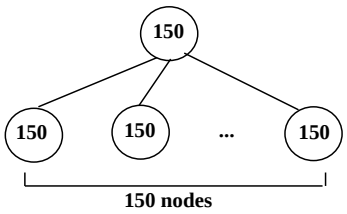

Figure 2: Search space for Matrix 150, in Aurora

For this application, the speedups were not affected by the cache coherence protocols. This is mainly because of the regular nature of this kind of problem.

It is interesting to notice that while the parallel C program has a high miss rate with a smaller total of references to shared memory (~6M for one processor), Aurora with a much larger total of references (~142M for one processor) has a very low miss rate running this application. This is shown in figure 3. This indicates that Aurora running the matrix application has better locality than C. This is mainly due to the design of the abstract machine used for Aurora that is based on the WAM [33, 1] and it is optimised for locality. However, although this can be considered an advantage, it pays off because the high level of the abstract machine requires many more references to execute the emulated code. While C MIPS code is directly executed by the simulator, the matrix program is emulated by Aurora that is executed by the simulator. We pay for having a high level language.

Most of the C misses come from evictions misses. This happens because of the memory organisation

(one-way associative) and cache size (64Kbytes) that does not fit one entire matrix.

Once more, we can observe that the miss rates were not affected by any protocol.

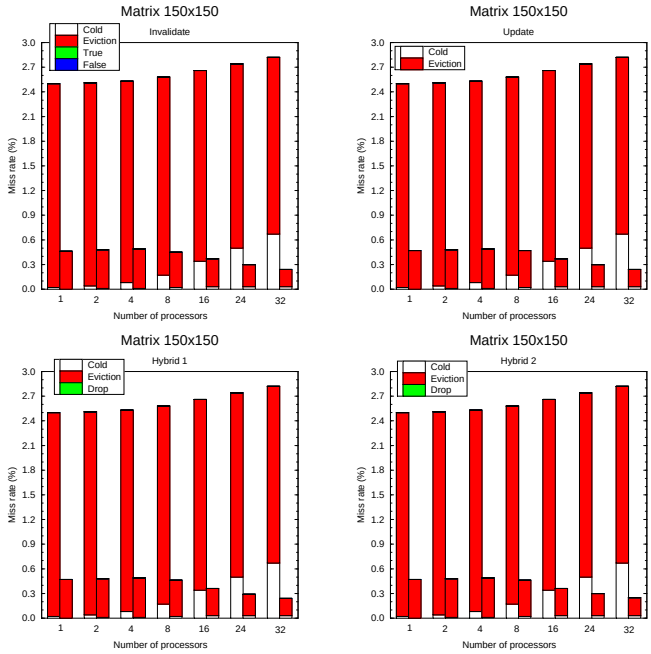

Figure 3: Matrix miss rates, four protocols, C (left) and Aurora (right)

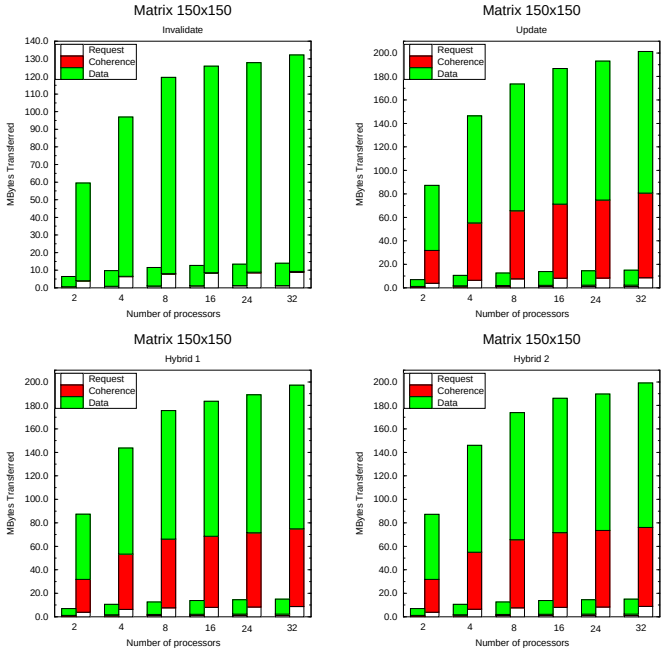

Figure 4: Matrix Network Traffic, four protocols, C (left) and Aurora (right)

Contrasting with the very low miss rates obtained for the matrix application, the network traffic for Aurora is much larger than for the parallel C matrix program. This can be observed in figure 4. This high network traffic for Aurora is easily explained by the high number of references to shared memory. As expected, the WI protocol generates less traffic than the WU protocol for the C and Aurora programs. The hybrid protocols contributed very little to an improvement in the WU performance. As this application is very regular, a change in protocol is not noticed.

The majority of network messages for WI are for data being exchanged while WU, Hyb1 and Hyb2 exhibits a much higher rate of coherence messages for Aurora when compared with the C network messages, but this is again due to the total references made by Aurora. It is interesting to note that the network traffic for Aurora grows slower as we increase the number of processors. This is observed for all protocols. The reason is that Aurora with few processors may create the rightmost choicepoints below the root tree later than when we use a larger number of processors. This makes the few workers compete for the same node to find work. Once we increase the number of processors, and the first leftmost choicepoint is exhausted, idle workers have chance to get work from the root node or from different choicepoints created by other processors at level 2. Therefore each worker that starts one alternative at the root node or at a choicepoint at level 2 can work independently on its own task without interfering with each other to steal work.

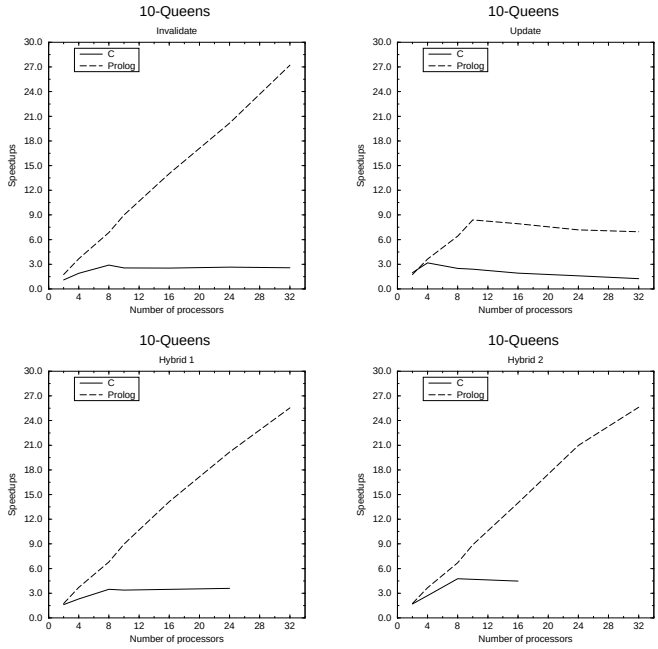

Figure 5: Queens speedups, four protocols

N-queens Figure 5 shows the speedups for the 10-queens problem. Because of the board size (10x10) and the way the parallel algorithm was implemented in C, we also run this application for 10 processors. As mentioned before, this problem is very difficult to parallelise efficiently in a procedural program. The search space for each board taken by a worker causes a great amount of load imbalance. In order to minimise load imbalance, the boards are created and put on a shared task queue by the parent process. But this optimisation was not enough to remove completely the load imbalance. Moreover, the boards allocations caused a very high rate of false sharing misses as can be easily observed in figure 6, for WI protocol. The false sharing caused a high rate of drop misses for the hybrid protocols. Hyb2 had a lower drop miss rate because was less eager to do self-invalidations. Depending on the way processes take the boards from the queue, the memory access patterns can cause lower or higher miss rates.

This application can only sustain parallelism in C up to 10 processors because of the board size and the way the work is distributed among the processors. The C parallel implementation uses the algorithm as explained in section 4.

Hyb2 seems to yield the best performance for the parallel C program. WI is not so good as WU, Hyb1 and Hyb2 for the C parallel program, because WI generates a higher miss rate. The miss rates for WU for the parallel C program are so low that do not fit on figure 6.

An interesting point to mention is that although hybrid protocols produced, in general, better performance than WU for the parallel C program, they also produced higher miss rates than WU. This higher miss rate was counterbalanced by a reduction in the number of update messages, as can be observed in figure 7.

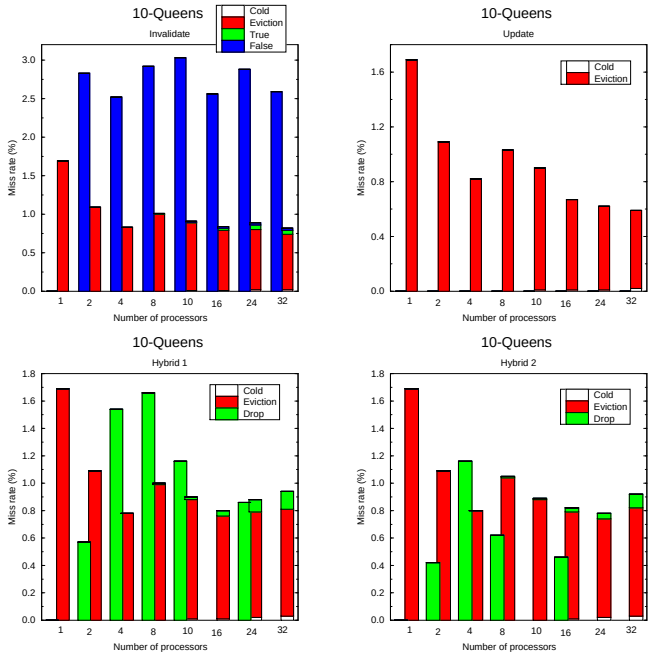

Figure 6: Queens miss rates, four protocols, C (left) and Aurora (right)

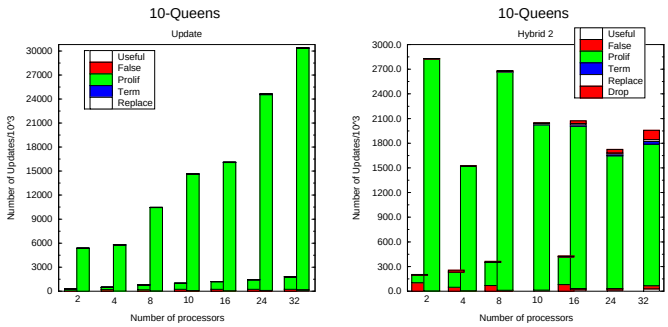

Figure 7: Queens update messages, two protocols, C (left) and Aurora (right)

Aurora manages to keep performance achieving efficiency of 85% with the best coherence protocol, invalidate, on 32 processors.

For this application, WI seems to yield the best overall performance for the Aurora version. This is mainly due to the high network traffic caused by WU for this application in Aurora. This can be seen in figure 8.

The WU protocol exhibits very bad performance for this application in Aurora from 10 processors on. The main responsible for this loss in performance is the increase in number of coherence messages. The network traffic figure shows that the coherence traffic is in average almost 62 times larger for WU than for WI. This causes a high network contention that deteriorates performance.

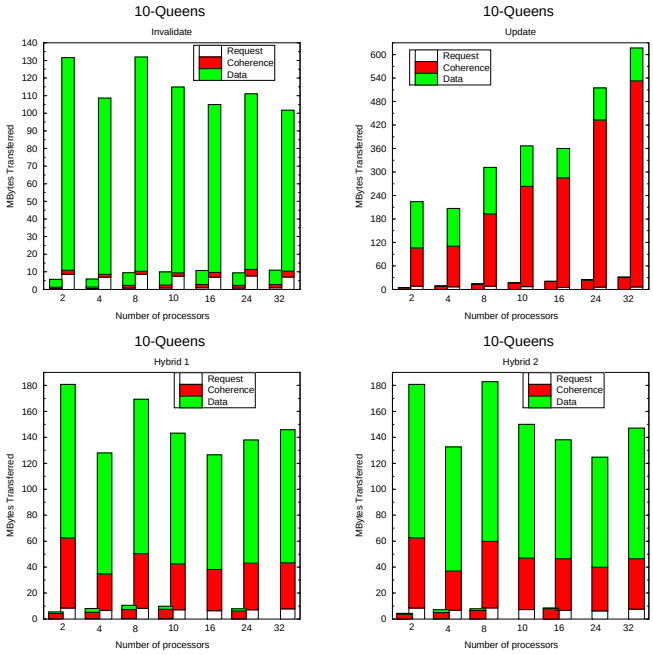

Figure 8: Queens Network Traffic, four protocols, C (left) and Aurora (right)

Due to limitations on the simulator, we could not obtain some simulated results: Hyb1, parallel C program for 16 and 32 processors. Hyb2, parallel C program for 10, 24 and 32 processors.

TSP Figure 9 shows the speedup curves for the tsp benchmark. The performance for both the parallel C program and the Aurora program was very bad.

The Aurora program is implemented by asserting and retracting the tour cost at each new search for a new tour. We use the built-in predicates `assert/1` and `retract/1` to add and remove the fact representing the tour cost from the database. The workers need to wait to be leftmost on the tree to execute these built-in predicates. This causes a serialisation on the parallel execution that degrades performance as we increase the number of processors. We confirmed that by visualising the parallel execution tree.

The parallel C program shows a very high rate of false and true misses for WI and a high ratio of drop misses for the Hyb1 and Hyb2 protocols. This can be observed in figure 10. This happens because the shared data structures can be allocated in a way that we can have more than one tour in the same cache block that causes false sharing, and shared counters, flags and locks that cause true sharing.

Database Figure 11 shows the speedups for the database application. The parallel C program scales up well while the Prolog program has very bad performance. WI seems to produce slightly better results for the Aurora version if we compare the absolute numbers (we could not run the Aurora version for 24 and 32 processors for WU, Hyb1 and Hyb2 due to some limitations on the simulator). For the parallel C program, any protocol shows good performance.

The parallel C program is implemented in a way that each processor executes one query at a time. There is no shared memory, excluding the barrier. At 24 processors, the speedup is similar to the speedup for 16 processors, because the number of queries is not a multiple of the number of processors.

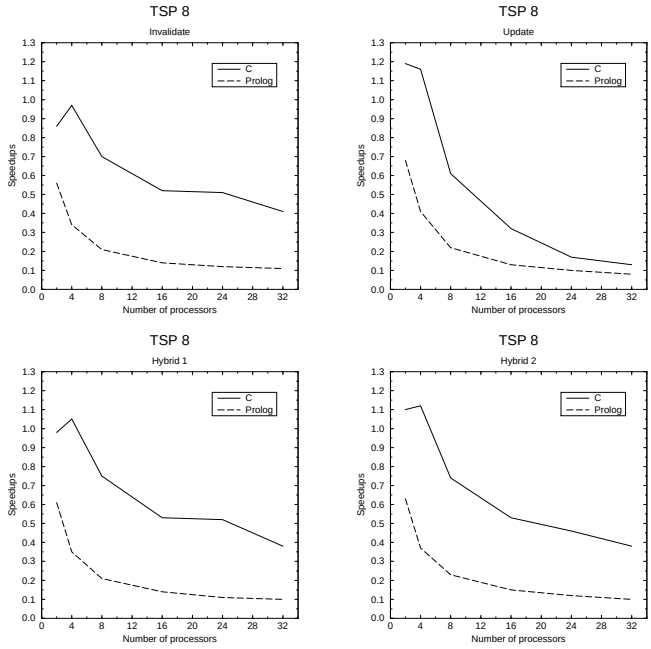

Figure 9: TSP SPEEDUPS, FOUR PROTOCOLS

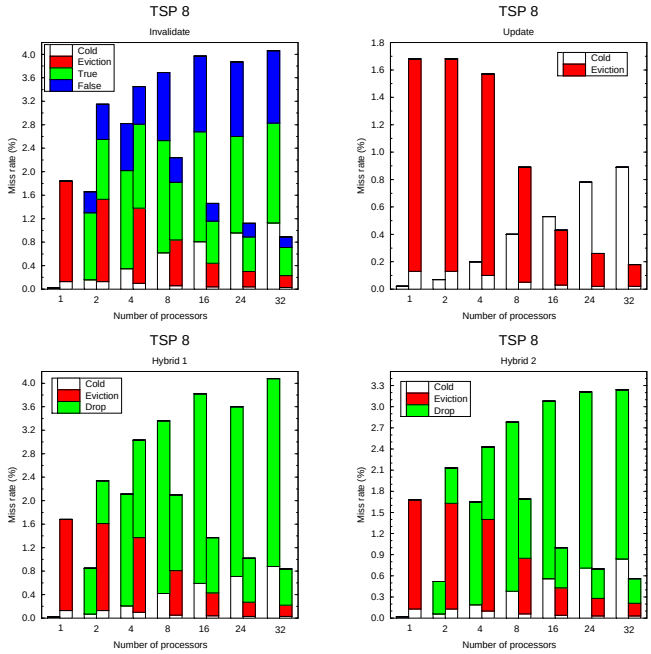

Figure 10: TSP MISS RATES, FOUR PROTOCOLS, C (LEFT) AND AURORA (RIGHT)

In figure 12 the parallel C program exhibits only cold misses that effectively represent all misses on this application. As expected, the network traffic is limited only to the exchange of the barrier value in the end of the program. It does not appear in figure 13 because the values are shown in Megabytes.

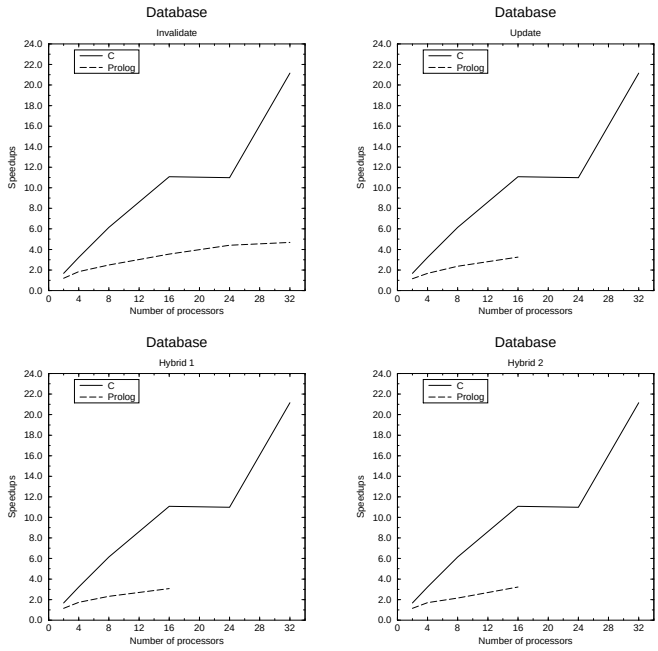

Figure 11: Database speedups, four protocols

The Aurora version of the database generates a very irregular execution tree with four public choicepoints at the second level of the tree. The number of alternatives in each of these choicepoints varies. Therefore the work distribution can cause some load imbalance or contention on nodes that have a big number of alternatives, but very fine grained work below each alternative. In fact, this is the reason for the very low speedup exhibited by this application in Aurora. Displaying the graphical representation of the parallel execution tree helped to identify this problem [23]. Modifications to the scheduler or utilisation of compile-time granularity information could improve performance.

In figure 12 we can see that Aurora has an acceptable miss rate not exceeding 8% among the four protocols, even having Aurora a very high number of references to shared memory. This indicates once more that Aurora maintains reference locality. Most misses are due to eviction misses.

5 Conclusion and Future Work

We used a detailed execution-driven simulation to thoroughly evaluate the performance of parallel C programs and of Aurora, a parallel logic programming system that exploits or-parallelism.

Our results indicate that cache coherence protocols do not significantly affect regular applications while irregular ones can benefit from hybrid or invalidate protocols. Our results also show that parallel logic programming systems written for shared memory machines can achieve good scalability on modern architectures depending on the application, but at the cost of high time and space complexity when compared to conventional C programs.

Results also indicate that optimisations such as improvements on the abstract engine, data organisation and scheduling can make parallel logic programming systems competitive and more attractive than parallel C systems for some applications, specially if we consider that parallel logic programming systems exploit implicit parallelism.

Our results also show that although Aurora has a huge number of references to shared memory, in general, it manages to maintain good reference locality while parallel C programs with very low number of references need a good programming style for the kind of architecture we are studying.

The main problems encountered on our experiments are related to scheduling issues or data layout. Our next steps will be to use different scheduling strategies and data restructuring both for the parallel C

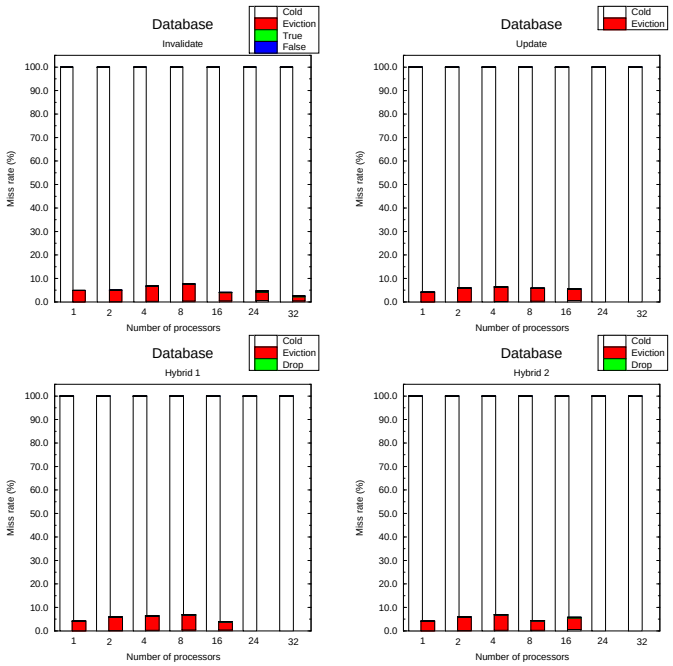

Figure 12: Database miss rates, four protocols, C (left) and Aurora (right)

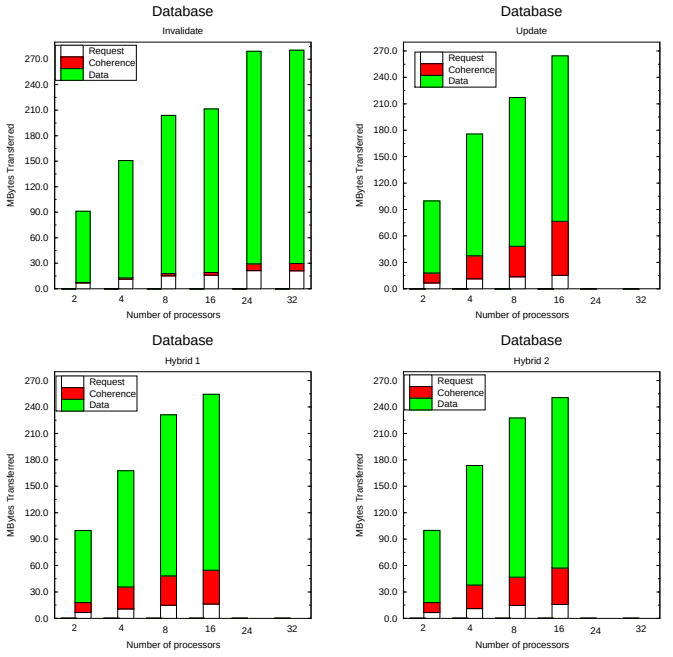

Figure 13: Database Network Traffic, four protocols, C (left) and Aurora (right)

programs and for Aurora in order to improve performance. Another step will be to evaluate the impact of different architectural parameters such as cache sizes, cache block sizes and network bandwidth sizes on our applications. Another step will be to evaluate real Prolog and C symbolic applications.

Acknowledgments

We would like to thank the Brazilian Research Councils (CNPq and CAPES) for supporting this research. Vanusa would like to thank her bosses Sergio Abramovitch and Volnei Marques da Costa for allowing her to carry out this research work.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction*. MIT Press, 1991.

[2] 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.

[3] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.

[4] Anthony Beaumont and David H. D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 135–149. MIT Press, June 1993.

[5] R. Bianchini and L. I. Kontothanassis. Algorithms for categorizing multiprocessor communication under invalidate and update-based coherence protocols. In *Proceedings of the 28th Annual Simulation Symposium*, April 1995.

[6] Vanusa Menditi Calegario and Inês de Castro Dutra. Performance Comparison between Conventional and Logic Programming Systems. Technical Report ES-478/98, COPPE/Systems Engineering and Computer Science, Setembro 1998.

[7] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1997. Release 3#6.

[8] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th ISCA*, pages 88–97, May 1993.

[9] Vincent W. Freeh. A Comparison of Implicit and Explicit Parallel Programming. *Journal of Parallel and Distributed Computing*, (34):50–65, 1996.

[10] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.

[11] Gopal Gupta, Enrico Pontelli, and Manuel Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

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

[13] Markus Hitz and Erich Kaltofen, editors. *Proceedings of the Second International Symposium on Parallel Symbolic Computation, PASCO'97*, July 1997.

[14] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings of the 17th ISCA*, pages 148–159, May 1990.

[16] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[17] 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.

[18] Kim Marriot and Peter J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[19] E. M. McCreight. The Dragon Computer System, an Early Overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[20] S. Raina, D. H. D. Warren, and J. Cownie. Parallel Prolog on a Scalable Multiprocessor. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 27–44. Wiley, 1992.

[21] V. Santos Costa, Bianchini, and I. C. Dutra. Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation, PASCO'97 [13]*, pages 58–67, July 1997.

[22] V. Santos Costa, R. Bianchini, and I. C. Dutra. Evaluating the impact of coherence protocols on parallel logic programming systems. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 376–381, 1997. Also available as technical report ES-389/96, COPPE/Systems Engineering, May, 1996.

[23] V. Santos Costa, N. F. Fonseca, and I. C. Dutra. VisAll: A Universal Tool to Visualise the Parallel Execution of Logic Programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, Jun 1998.

[24] V. Santos Costa and Bianchini R. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, Sep 1998.

[25] Márcio G. Silva, Inês C. Dutra, Ricardo Bianchini, and Vítor Santos Costa. The Influence of Computer Architectural Parameters on Parallel Logic Programming Systems. Technical report, January 1999. Also available as Technical Report ES/477-98, COPPE Systems Engineering, Sep/98.

[26] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. 20(1):5–44.

[27] R. Sosic and J. Gu. Fast Search Algorithms for the N-Queens Problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1572–1576, Nov/Dec 1991.

[28] Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart. The alpha demonstration unit: A high-performance multiprocessor for software and chip development. *Digital Technical Journal*, 4(4):51–65, 1992.

[29] Evan Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[30] P. Van Roy and A. M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, 25(1):54–68, January 1992.

[31] J. E. Veenstra and R. J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

[32] J. E. Veenstra and R. J. Fowler. Mint tutorial and user manual. Technical report, University of Rochester, Computer Science Department, 1994.

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

[34] David H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 International Logic Programming Symposium*, pages 92–102, 1987.

[35] Rong Yang, Vítor Santos Costa, and David H. D. Warren. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.