Performance of Sparse Binding Arrays for Or-Parallelism

Vítor Santos Costa, Manuel Eduardo Correia, and Fernando Silva
{vsc,mcc,fds}@ncc.up.pt
LIACC, Universidade do Porto
Rua do Campo Alegre, 823
4150 Porto, Portugal

June 19, 1996

Abstract

One important problem in the design of novel logic programming systems is the support of several forms of implicit parallelism. A new binding model, the Sparse Binding Array (SBA), has been proposed for the efficient and simplified integration of Independent-And, Determinate-And and Or-parallelism. In this paper we report on the use of this model for pure Or-parallelism.

The work discusses the major implementation issues in supporting this binding model for pure Or-parallelism. We show that an implementation based on this Binding model is more efficient then the original Aurora using the traditional Binding Array model [16]. Moreover, we explain how the notion of a variable *level* can be used to reduce overheads of the Or-parallel system. Our results in supporting pure or-parallelism show that the approach is very promising for combined parallel systems.

1 Introduction

One of the advantages of logic programming is the fact that one can exploit *implicit* parallelism in logic programs. The most relevant forms of parallelism are Or-parallelism, as exploited in the Aurora [11, 4] and Muse [1, 10] systems, Independent And-parallelism, as exploited in &-Prolog [9], and dependent And-parallelism, as exploited in the committed-choice languages or in Andorra-I [14]. Most recently, research has been concentrated on integrating the different forms of parallelism in a single combined system.

Or-parallelism is one of the main forms of parallelism to include in a combined system. But before this can be achieved two main problems must be addressed : scheduling and variables bindings representation. The scheduling problem can be addressed rather orthogonally to the other forms of parallelism. In contrast, bindings representation affects all forms of parallelism.

The problem in binding a variable arises because each may have several different bindings in different or-branches. Quite a few very different approaches [7] have been presented to tackle this problem. Two successful ones are copying, as used in Muse [1], and binding arrays, as used in Aurora.

In the copying approach, each worker maintains its own copy of the path in the search tree it is exploring. Whenever there is some work to be shared, the worker that is looking for work copies the entire path up to the current node of the worker that it is sharing. Sharing between workers only happens through an auxiliary data structure associated with choice-points.

In the binding array approach the work stacks are shared. To obtain an efficient access, each worker maintains a private data structure, the binding array. The binding array gives the worker quick access to the binding of a variable. It is implemented as an array, indexed by the number of variables that have been created in the current branch. This number is also stored in the variable itself which gives constant-time access to private variable bindings.

Both schemes have problems when the system is extended with And-parallelism. In the case of copying, the main problem is what to copy because workers working in And-parallel will be at different points in the search tree. The ACE system addresses this problem for a combination of Or-parallelism with Independent And-parallelism. The same happens for the Penny [12] parallel implementation for AKL.

In the case of binding arrays the main problem are workers working in And-parallel, hence that share the same Team Binding Array, that may want to place their variables in the same binding array positions. A solution to the binding array problem is the PBA [8], and the SPBA [5] data structures. These data structures use indirection to address the binding array management problem. They present a very general solution to the problem, but introduce an extra dereferencing layer in the binding array scheme.

In this paper we introduce a novel data structure, the *Sparse Binding Array*, or SBA. The Sparse Binding Array simplifies the traditional binding array and naturally allow the representation of Independent and-computations running in Or-parallel. A variable's binding array entry is not associated with a relative position in the search tree branch where it was created, as in the BA approach, but with the variable's memory address.

The paper is organized as follows. We first describe the sparse binding array data structure, comparing it with the traditional binding array. We next discuss some important implementation issues and explain how levels are used to obtain variable age. We then discuss in detail how the Aurora system was extended to support the SBA data-structure, and give a preliminary performance analysis for a standard set of benchmarks. We finish with our conclusions and proposals for further work.

2 The Sparse Binding Array

We first define some terminology. A worker is a processing agent, usually a process running on a separate processor. A team is a group of workers that share the same bindings, that is, that are running in And-parallel. We assume that Or-parallelism is exploited by having these teams running in Or-parallel, as in Andorra-I, or as in the C-tree model [6].

In our model, as in the original SRI model, the environment, choice-point and global stacks, trail and code segment are all shared between workers. Following the cactus stack concept, each worker has its area of work on the stacks, where he writes and from where other workers can only read. The exception to this rule is the choice-point stacks. To fetch Or-parallel work, workers will need to change shared choice-points. In this case, access is synchronized.

Each team keeps a private set of bindings for shared variables. Therefore these data structures are shared between workers in a team, and are not shared between teams. We say that a team's binding array *shadows* the shared variables. The key idea in the binding array concept is that for each variable in the stacks there is a single, uniquely identified, binding array slot that is the same for any team. This allows for fast access to a binding from a variable.

To implement the concept, the SRI model "binds" each variable X to the number of variables N between X and the root of the computation tree. To find out the private binding of the variable X at $SHARED\_PTR$, it is sufficient to search for the following value:

$$PRIVATE\_VALUE = BA\_BASE[*(SHARED\_PTR)]$$

where $PRIVATE\_VALUE$ represents the value we are looking for, BA_BASE represents the origin of the corresponding binding array, and $SHARED\_PTR$ represents the position of the shared variable, with $*(SHARED\_PTR)$ giving the value X.

In the Binding Array approach each new variable is initialized with an offset that actually serves as a pointer to the binding array. Also notice that binding array cells are allocated sequentially, that is, the entry for V_{i+1} immediately follows the entry for V_i.

Unfortunately the scheme breaks down when we do not know the number of variables between a variable X and the root of the tree. This is the case when we have And-parallelism.

To address this problem, the *Sparse Binding Array* extends the original binding array with the further restriction that this variable position must be *unique*, that is, no other currently accessible variable may have the same binding array offset. In this way, if several workers in a team create variables at the same time, they are guaranteed never to use the same binding array slot.

To implement this concept, the SBA uses a direct mapping from the global stack to the global binding array, and from the local stack to the local binding array. The mapping is from pointers to a shared stack to pointers to the corresponding binding array and is as follows:

$$SHARED\_PTR \Rightarrow SHARED\_PTR + (BA\_BASE - STACK\_BASE)$$

Thus, the private value can be obtained as follows:

$$PRIVATE\_VALUE = SHARED\_PTR[BA\_BASE - STACK\_BASE]$$

where $STACK\_BASE$ represents the base of the corresponding stack. Again we have a constant-time operation, giving fast access. This access can be made quite fast if we know before-hand $STACK\_BASE$ and BA_BASE. In fact, by using a shared memory interface such as `MMAP` or `SHM` we may indeed know the two values at compile-time.

Note also that for each variable in the local and global stacks, we are creating at least as many "virtual" copies in the SBA. We have used the word virtual because most of this memory will never be actually used, only requested. As we shall discuss in more detail later, modern Operating Systems only actually allocate memory that is being used.

3 Sparse Binding Array Implementation Issues

In this section we discuss in detail two important issues in the SBA. First, we discuss the problem of variable representation in the SBA, and next we discuss in more detail the problem of memory allocation. It is clear that some space allocated for the SBA will be wasted. This problem can be offset by the advantages of a simpler memory management scheme and by the support for Independent And-parallelism.

3.1 Variable Representation in the SBA

If we use the SBA as the underlying system for the management of environment space what should be the value of a variable cell?

In the BA scheme a cell's value is an index into the binding array. In the PBA[8] scheme the cell value is required to determine the location of its private environment value. In the SBA it is the address of the cell that is used.

Both the BA and the PBA carefully maintain the cell value as an indication of variable age. This is necessary because in a parallel implementation we cannot just compare addresses to obtain ages. In the SBA scheme, we do not need to use indexes, but we still require variable age maintenance to avoid conditional bindings and to trim the trail when we cut.

We could have preserved the Aurora scheme. Aurora uses two counters, one for local and other for global variables.

A first possible simplification is to use a single counter. A more interesting simplification is to avoid having a counter, and instead to use a *variable level* representing the number of choice points above it in the execution tree. If we carefully design our systems to always allocate memory in such way that any private segment of branch will always be composed of memory chunks in ascending order (as it is the case in Aurora) then variable age can be simply determined by the lexicographic order of the pair **(variable_level, variable_address)** .

The main advantage is that instead of updating an index each time a variable is created, the level will be updated only when a choice point is allocated.

A second problem we needed to address with this scheme was the cut. When we have cut, several choice-points are discarded. If we are at level O, cut to level M, and bind a variable created at level N, where $M < N < O$, a binding to the variable is still deterministic. The obvious solution would be to reset the counter to M. The problem is that the next time we create a choice-point, its level would be $M + 1$, hence every variable already created at levels between $M + 1 < O$ would be considered to be bound unconditionally, when in fact they can only be bound conditionally.

The solution to this problem is to use two counters. One counter, `level-top` represents the level we have reached so far, whereas `level-uncond` represents the level for unconditional variables.

Private operations with levels are as follows:

```
new-var : *SHARED_ADRESS = level-top;

conditional : (*SHARED_ADRESS < level-uncond)

try : B→level = level-uncond = level-top = level-top+1;

retry : level-top = level-uncond = B→level;

trust : level-top = B→level-1; level-uncond = B→parent→level;

cut : level-uncond = ■B→level;
```

Note that the approach is similar to the way Aurora manipulates the binding array counters. As in Aurora, the major problem arises when we have shared and private parts of the search tree. We discuss the problems in further detail in section 4.3

3.2 Trusting the underlying Operating System

One of our motivations in designing the SBA was the improvements in Operating System support for memory allocation. Modern operating systems support the direct mapping of files in shared memory. Page frames for this mapping are allocated lazily (each time they are referenced) and later written to the mapped file in disk. Consequently while virtually we may have requested a big chunk of address space the operating system only allocates a much smaller number of page-frames and supporting swap space.

Note that the operating system takes automatic care of the dynamic allocation of memory and mapped file disk blocks each time a reference is made to a non allocated page. Current file operating system implementations are also optimized to only use a small number of real physical disk blocks . This means that the actual amount of physical memory *we need is not what we ask for, but what we use.*

The question for the SBA is therefore of how much more memory we use in the Sparse Binding Array. The answer is that the pages we will need in the Or-parallel implementation are the pages where variables were created, and later on referenced. These pages consist of variables plus environment control information, for the local stack, and variables plus compound terms constructors and atoms, for the heap.

Notice also that when these system pages are no longer in use the operating system automatically pages them out of memory. As a consequence the only thing that the system implementor has to do when using SBA is to map memory space and leave to the underlying operating system the task of its efficient maintenance.

Finally, we must not forget the fact that we are using the WAM [15] as the underlying execution engine. As it is well known, the WAM has very good memory locality properties. The SBA inherits these properties.

4 The Or-Parallel Implementation of the SBA

As a first step towards a combined SBA based and-or system we have studied SBA's implementation in Aurora. We concluded that the necessary changes were reasonably small as can be seen in detail with the discussion that follows.

4.1 Memory Allocation

We preserved Aurora's memory allocation for the shared stacks but investigated two approaches for the SBA implementation in Solaris 2.4 .

MMAP: a large chunk of memory was allocated for the SBA. Unfortunately in our experience we found out the MMAP routines had severe problems. There were wide variations of performance between runs. Also, the MMAP best performance was obtained when we actually allocated all the memory first. If we did not ask for the memory immediately, there would be a severe degradation in performance.

MALLOC: the alternative was to use `malloc` to allocate space for the binding array. Surprisingly this gave much more stable performance in our platform, and has since been used in the benchmarks.

The actual space we ask for the SBA "shadows" all the stacks. These include code space, choicepoint stack, and trail, which need to be shadowed. In practice this should not matter, as the space is only *virtually* allocated.

In any case we can keep the offset $BA\_BASE - STACK\_BASE$ fixed and known at compile-time. In order to save space, Aurora does not have a fixed BA_BASE value [4], which results in worse performance versus the SBA system.

4.2 Engine in Private Region

Most of the time the Aurora engine will be executing private work. As we explained, execution in the SBA model could follow the exact execution in Aurora, the only difference being the macros for variable access.

After introducing the "level" scheme, we had to make changes to other operations, such as choicepoint manipulation and `cut`. These changes follow the principles presented in section 3.1. To support shallow backtracking [3] the actual implementation for Aurora is slightly more complex than in a traditional WAM [15].

A problem arises with the implementation of tail recursion in Aurora. Aurora keeps a free variable in every environment in order to calculate the age for the environment. In order to recover space it tests whether this age is younger than the age for the last choicepoint. The SBA implementation uses the same approach, giving each environment an extra slot with its age. Environments can be recovered if they are younger or if they have the same age as `level-uncond`.

Whenever it creates a choice-point Aurora verifies if there is an overflow of the binding array. This is not done within SBA thus accelerating choice-point creation.

4.3 Parallel Work

Parallel Work happens in Aurora when the engine enters a shared area of the search-tree, or when the engine calls a *scheduler*.

Work in the shared area can be: moving up the tree to backtrack, moving down the tree to select new work, and cutting shared work. Using sparse binding arrays does not force any changes to this scheme. Problems only arise with the new level-based scheme. In this case it is necessary to reset levels whenever we backtrack to shared areas of the tree, or whenever we create new work. Note that when creating work we never perform `trust` in the shared area because we cannot be sure someone else is exploring an alternative. The choice-point has been created before, hence the operation is always a `retry`. In Aurora the `retry` is always followed by creating an embryonic choice-point. These rules had to be followed very carefully.

5 Initial Performance Evaluation

We have measured the timings and speedups obtained both from Aurora with SBA and Aurora with Binding Arrays (table 1 and table 2). We have used an 8-CPU SparcCenter 2000 with 256MB of main memory, 1MB cache per CPU, running Solaris 2.4. The system was in multi-user mode. Both versions of Aurora were compiled with `gcc -O2`. We used the standard Aurora benchmark set. The scheduler we use is the distribution release of the Bristol scheduler [2], as ported to the Sparc [13].

With one worker the SBA is between 10 and 15% faster then Aurora. Increasing the number of workers, the SBA generally outperforms Aurora, although for higher number of workers Aurora comes close and sometimes actually performs better then SBA.

We believe that the better sequential performance is explained by the advantages of the SBA: easy calcuıatiun of the BA address, and the use of levels for age comparison of variables.

As regards speedups, we think Aurora can achieve better speedups because:

- Aurora has worst one-worker performance, and therefore has coarser task-granularity. This means that scheduling overheads will be less significant.
- Task-switching in the SBA is more expensive because the page working set is bigger.

We expect the speedups to improve by using the newer releases of the Bristol scheduler.

Finally note that these results are still preliminary. Further tests and optimizations shall be carried out.

	num_workers						
Goals [•Times]	1	2	3	4	5	6	7
parse1 •200	1.62	1.14(1.42)	1.13(1.44)	1.25(1.30)	1.25(1.30)	1.26(1.29)	1.27(1.27)
parse2 •200	6.36	3.64(1.74)	3.20(1.99)	2.83(2.24)	2.71(2.35)	2.61(2.43)	2.54(2.50)
parse3 •200	1.32	1.03(1.29)	1.04(1.28)	1.13(1.17)	1.16(1.15)	1.20(1.11)	1.24(1.07)
parse4 •50	5.81	2.97(1.96)	2.28(2.55)	1.88(3.09)	1.69(3.44)	1.55(3.76)	1.46(3.98)
parse5 •10	3.86	2.04(1.89)	1.45(2.66)	1.20(3.23)	1.04(3.72)	0.92(4.20)	0.87(4.44)
db4 •100	2.52	1.58(1.60)	1.19(2.12)	1.02(2.48)	0.92(2.73)	0.90(2.82)	0.86(2.94)
db5 •100	3.27	2.03(1.61)	1.45(2.25)	1.23(2.65)	1.14(2.88)	1.07(3.06)	1.03(3.17)
farmer •1000	2.96	2.48(1.19)	3.09(0.96)	3.61(0.82)	3.96(0.75)	4.21(0.70)	4.51(0.66)
house •200	4.09	2.69(1.52)	2.38(1.72)	2.20(1.85)	2.10(1.94)	2.04(2.01)	1.98(2.06)
8-queens1 •10	7.00	3.58(1.95)	2.44(2.87)	1.85(3.79)	1.52(4.61)	1.29(5.43)	1.13(6.18)
8-queens2 •10	17.99	8.89(2.02)	5.98(3.01)	4.50(4.00)	3.70(4.86)	3.11(5.79)	N/A
tina •10	14.68	8.20(1.79)	5.48(2.68)	4.38(3.35)	3.62(4.05)	N/A	N/A
sm2 •100	10.03	5.35(1.88)	4.05(2.48)	3.10(3.23)	2.63(3.81)	2.33(4.30)	2.10(4.77)
AVERAGE		(1.68)	(2.15)	(2.56)	(2.89)	(3.07)	(3.00)

Table 1: Performance of SBA on a SparcCenter 2000

	num_workers						
Goals [•Times]	1	2	3	4	5	6	7
parse1 •200	1.86	1.17(1.59)	1.12(1.67)	1.18(1.58)	1.20(1.55)	1.21(1.54)	1.26(1.48)
parse2 •200	7.00	3.78(1.85)	3.07(2.28)	2.73(2.56)	2.61(2.68)	2.56(2.73)	2.62(2.68)
parse3 •200	1.49	1.10(1.36)	1.11(1.34)	1.18(1.26)	1.19(1.25)	1.24(1.20)	1.30(1.15)
parse4 •50	6.07	3.25(1.87)	2.44(2.49)	1.96(3.10)	1.76(3.45)	1.63(3.72)	1.54(3.96)
parse5 •10	3.99	2.13(1.87)	1.50(2.66)	1.24(3.22)	1.05(3.78)	0.94(4.22)	0.87(4.59)
db4 •100	2.57	1.50(1.71)	1.15(2.24)	0.98(2.63)	0.89(2.90)	0.86(3.00)	0.85(3.05)
db5 •100	3.15	1.82(1.73)	1.32(2.39)	1.18(2.68)	1.08(2.91)	1.04(3.04)	0.99(3.19)
farmer •1000	3.24	2.36(1.37)	3.07(1.06)	3.63(0.89)	3.95(0.82)	4.16(0.78)	4.49(0.72)
house •200	4.36	2.63(1.66)	2.37(1.84)	2.18(2.00)	2.09(2.09)	1.99(2.19)	1.97(2.21)
8-queens1 •10	8.14	3.84(2.12)	2.59(3.15)	2.06(3.96)	1.66(4.90)	1.40(5.83)	1.19(6.84)
8-queens2 •10	20.06	9.75(2.06)	6.37(3.15)	4.79(4.19)	3.89(5.16)	3.32(6.04)	2.92(6.88)
tina •10	15.57	7.95(1.96)	5.51(2.83)	4.21(3.70)	3.47(4.49)	2.94(5.30)	2.62(5.94)
sm2 •100	10.24	5.42(1.89)	3.75(2.73)	3.03(3.38)	2.51(4.07)	2.24(4.58)	2.07(4.94)
AVERAGE		(1.77)	(2.29)	(2.70)	(3.08)	(3.40)	(3.66)

Table 2: Performance of Aurora on a SparcCenter 2000

6 Conclusions

We presented the Sparse Binding Array, an efficient environment representation scheme designed to support both Independent And- and Or-parallelism. The scheme can be easily ported to traditional Binding Array based systems. In this paper we show that the ensuing implementation can actually outperform the corresponding Binding Array based implementation, at least in sequential execution.

We believe this approach to be very promising for combined parallel systems, as the binding model directly supports Independent And-parallelism and Determinate And-parallelism

Acknowledgments

The authors would like to acknowledge and thank for the contribution and support that Gopal Gupta, Enrico Pontelli, K. Shen, and Manuel V. Hermenegildo gave to this work. This work has been supported by the JNICT PROLOPPE project and by the NATO MAPLE project. Manuel Eduardo Correia thanks JNICT for supporting his research. Vítor Santos Costa thanks the Universidade Federal do Rio de Janeiro for supporting his current research.

References

[1] K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] A. Beaumont, S. M. Raman, P. Szeredi, and D. H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991.

[3] M. Carlsson. On the efficiency of optimised shallow backtracking in Compiled Prolog. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 3–15. MIT Press, June 1989.

[4] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. SICS Dissertation Series 02, The Royal Institute of Technology, 1990.

[5] G. Gupta, V. S. Costa, and E. Pontelli. Shared Paged Binding Array: A Universal Datastructure for Parallel Logic Programming. In *Proceedings of the Twelveth International Conference on Logic Programming*, to appear.

[6] G. Gupta, M. Hermenegildo, and V. S. Costa. And-Or Parallel Prolog: A Recomputation based Approach. *New Generation Computing*, 11(3,4):770–782, 1993.

[7] G. Gupta and B. Jayaraman. On Criteria for Or-Parallel Execution Models of Logic Programs. In *Proceedings of the North American Conference on Logic Programming*, pages 604–623. MIT Press, October 1989.

[8] G. Gupta and V. Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer-Verlag, June 1992.

[9] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[10] R. Karlsson. *A High Performance OR-parallel Prolog System.* SICS Dissertation Series 07, The Royal Institute of Technology, 1991.

[11] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*, pages 819–830. ICOT, Tokyo, Japan, Nov. 1988.

[12] J. Montelius and K. A. M. Ali. An And/Or-Parallel Implementation of AKL. Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments, CIS-94-04, University of Oregon, Mar. 1994.

[13] V. Santos Costa, M. E. Correia, and F. Silva. Aurora and Friends on the Sun (Extended Abstract). In *2nd COMPULOG NET Workshop on Parallelism and Implementation Technologies, Madrid,*, 1994.

[14] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

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

[16] D. 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.