Using Sparse Binding Arrays for Or-parallelism

(Extended Abstract)

Manuel Eduardo Correia, Vítor Santos Costa, and Fernando Silva*

{*mcc,vsc,fds*}*@ncc.up.pt*

LIACC, Universidade do Porto
Rua do Campo Alegre, 823
4150 Porto, Portugal

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 [9, 3] and Muse [1, 8] systems, independent And-parallelism, as exploited in &-Prolog [7], and dependent And-parallelism, as exploited in the committed-choice languages or in Andorra-I [12]. The main goal of our research is to design systems that will both exploit Or-parallelism and independent And-parallelism, taking advantage of available research.

A crucial consideration in the design of logic programming systems that exploit Or-parallelism is how to efficiently represent the multiple environments that arise at run-time. The same variable may have several different bindings in different Or-branches. Many successful solutions have been described in the literature. In our case, we were interested in one representation that would allow for **(i)** efficient representation, and for **(ii)** easy combination with Independent-And parallel execution. Following our previous research in the area, we were interested in the SPBA (Shared Paged Binding Array) [6]. This is a very flexible binding model, quite adequate for the proposed models of execution. Still during our implementation of the SPBA, it became clear to us that there would be advantages in considering a simpler binding scheme.

In what follows we describe a new binding model, the Sparse Binding Array (SBA), that allows for efficient integration of Independent And- and Or-parallelism. We will concentrate on its use for Or-parallelism. We report our implementation of this scheme over our previous port of the Aurora system to the SparcCenter2000 [11] running the Solaris operating system, and show that this new scheme can be, for Or-Parallelism only, more efficient then the pure Binding Array model [15] as used in Aurora.

2 The Sparse Binding Array Model

The key idea for the Sparse Binding Array is that each workers shadows in its private binding array all the shared memory that is being used in the parallel system. In the case of Aurora, for example, each worker

*Correspondence should be addressed to Manuel Eduardo Correia, Laboratório de Inteligência Artificial e Ciência de Computadores, Universidade do Porto, R. do Campo Alegre 823, 4150 Porto, Portugal; telephone: +351 2 6001672x114; fax: +351 2 6003654; email: mcc@ncc.up.pt

allocates a private local space of virtual memory whose size is virtually the same as the combined size of all shared memory that is being allocated by the system and used by all the workers.

Following [5], we shall assume that combined IAP/OR Parallel system will be implemented by having teams of workers. Workers within a team will work in And-parallel, whilst each team will work in Or-parallel with the other teams. In this case, the SBA will still mirror all the shared areas, but each sparse binding array will be shared by all the members of each team.

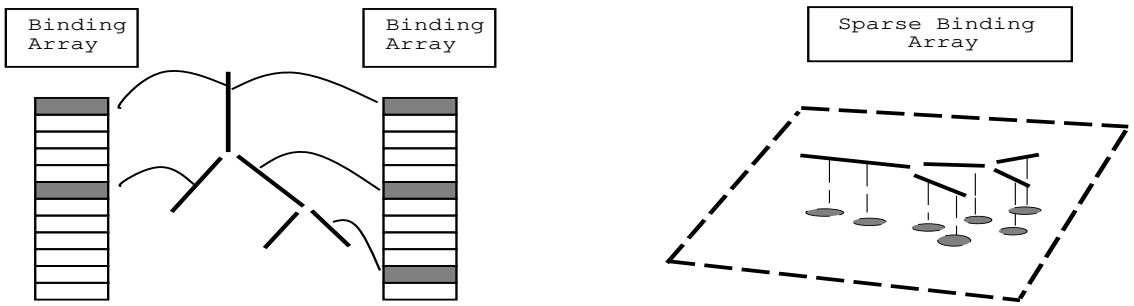

Figure 1: A graphical comparison between the BA classical aproach and SBA

In order to better understand what the SBA really is just think of it as a duplicate of the shared memory that is used by each Or-execution entity (a worker in the case of an Or-parallel system, a team in the case of an IAP/OR parallel system) to save its private environment in the simplest and fastest way. That is, by using the address of the cell whose value we intend to save in the private environment and adding up the fixed distance where the SBA have been allocated in memory we come up very quickly with the memory address of the private value for that environment cell.

Since each cell in the system has a unique address the private location for its local environment value is also automatically unique. Therefore, given a shared memory address we can very simply calculate its corresponding sparse binding array address by adding the offset from the start location of the SBA to the shared memory. If we always allocate the SBA at a predefined distance from the virtual address space, we can know this offset at compile-time, and thus have an extremely fast mapping.

In contrast, in the Binding Array scheme used in Aurora, or in the SPBA, used for a combination And (IAP, DAP) and Or parallelism [6], finding out the binding address is harder. In Aurora, one has to keep an offset to the binding array in the shared memory cell, whereas in the SPBA, one has to keep an offset in a page, obtained by looking up a page table.

It goes without saying that these schemes incur a non trivial cost (especially in systems where both forms of And and Or parallelism are being explored) in terms of execution time. Their advantage is that they minimise the use of binding array, by only allocating it when they need a new variable.

What we intend to prove in this work is that the cost of maintaining indexes and tables for the sole reason of having a unique location for a shared cell's value should be left to the underlying operating system wherever possible [1]. Why not use the facilities provided by the underlying operating system in the first place and at the same time simplify the design and implementation of parallel logic systems? We believe the SBA is the answer to this question.

[1] As have been pointed out by David H. D. Warren [6], the SPBA is akin to the memory management using paging in Operating Systems.

3 Trusting the underlying Operating System

One of our motivations in designing the SBA was Operating System support for memory allocation. Modern operating systems [2] [4] 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 initially only allocates a small number of page-frames and 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, the ones necessary to cope for the files data and its location inside a file. We are not going to discuss file systems implementation, but the point is that while the logical size of a file might be 1Gigabyte its real physical size in terms of the actual physical disk blocks used may easily be an order of magnitude smaller. This means that the actual amount of physical memory we need is not what we ask for, but *what we use* [3].

The question for the SBA is therefore 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 will be the pages were variables were created, and which consist of variables plus environment control information, for the local stack, and variables plus compound terms constructors and atoms, for the heap. If we assume that the heap and local stack consist mostly of variable cells, than our actual overhead in terms of space versus the traditional BA will be the remaining cells.

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 it and leave to operating system the task of efficiently maintain it.

Finally, we must not forget the fact that we are using the WAM [14] as the underlying execution engine. As it is well known, the WAM has very good memory locality properties [13]. The SBA inherits these properties. To understand why, note that a cell's private value location in the SBA is at a fixed distance from its address, and as a consequence the cell's locality is displaced (by the fixed distance) to the SBA.

4 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 SPBA it is pair of integers, which together give an offset. In both of these schemes the cell value is required to determine the location of it's private environment value. In the SBA it is the address of the cell that is used to determine it, so the value cell in itself is not required for this purpose.

Both the BA and the SPBA 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. We could have preserved the Aurora scheme, but we decided to simplify it by introducing the notion of levels.

We say that a *variable level* is 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 determined by the lexicographic order of the pair **(variable_level, variable_address)** .

[2] For example Solaris 2.4, any SVR4 and BSD4.3 compliant operating system or Linux.

[3] The mapped file can grow physically each time a new page is referenced in the system.

In terms of implementation, as have been done for Aurora [3], instead of updating an index each time a variable is created, the level will be updated only when a choice point is allocated.

The new scheme involves less bookkeeping than Aurora, because we do not need to update BA counters whenever we create variables, but is more expensive when we need to compare variable ages, as we may have to perform two comparisons.

5 Runtimes for SBA and Aurora

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 [11].

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 calculation 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 more binding array page frames will be touched.

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 optimisations 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

Goals [*Times]	num_workers						
	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 the sequential execution.

Work has started on integrating independent And-parallelism, as in &-Prolog [7] or in &-ACE [10] in the new system.

Acknowledgments

The authors would like to acknowledge and thank for the contribution and support that Gopal Gupta, Enrico Pontelli 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.

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

[4] B. Goodheart and J. Cox. *The Magic Garden Explained*. Prentice Hall, 1994.

[5] G. Gupta, M. Hermenegildo, E. Pontelli, and V. S. Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.

[6] 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.

[7] 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.

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

[9] 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.

[10] E. Pontelli, G. Gupta, and M. Hermenegildo. Implementation and performance of &ace: An independent and-parallel system. Technical report, NMSU, 1994.

[11] 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.

[12] 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.

[13] E. Tick. Prolog Memory-Referencing Behavior. Technical Report CSL-TR-85-281, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, September 1985.

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

[15] 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.