On Supporting Parallelism in a Logic Programming System

Vítor Santos Costa[1]

[1]CRACS and DCC-FCUP
Universidade do Porto
Portugal
vsc@dcc.fc.up.pt

Abstract. Logic Programming is a declarative approach to programming where one can specify a problem in a high-level fashion. Several major approaches to implicit and explicit parallelism have been proposed for logic programming in Prolog. But, arguably, the last few years have seen most interest in the explicit parallelization of Prolog.
With the advent of multi-core processors, parallelism is just available. One boring, but useful approach, is bag-of-tasks parallelism. We believe that the challenge facing parallel logic programming is to make all forms of parallelism as boring as possible. To do so, we propose some principles from our experience with previous work in Parallel Logic Programming, discuss how much a Prolog system needs to be adapted to support these principles, and present an application.

1 Introduction

Logic Programming is a declarative approach to programming where one can specify a problem in a high-level fashion. Arguably, Prolog is the most popular logic programming language. Early progress on Prolog compilation, leading to the WAM abstract machine [44], showed Prolog to be useful in a wide variety of practical applications. Prolog and Logic Programming have been widely used ever since, in a surprising large number of diverse applications.

The high-level nature of Logic Programming has made Prolog a natural target for parallelization. Several major approaches have been proposed. In *explicit parallelism* the programmer extends the language with a number of primitives that enable the creation and management of separate tasks. In *implicit parallelism* the Prolog system is largely responsible to detect and exploit the available parallelism [16].

Implicit and explicit parallelism have been well studied in logic programming, but, arguably, the last few years have seen most interest in explicit parallelization. The field of Inductive Logic Programming (ILP), within Machine Learning, has been an example motivation for some of this work. Systems such as April [14] and distributed versions of Aleph [22] were designed to run on clusters and apply MPI [4, 5] in a rather direct fashion. Thread libraries were used to parallelize Aleph in a conventional shared-memory machine. Machine Learning in general tends to generate computationally demanding tasks, and ILP is particular is highly computationally demanding. In order to support this need, Prolog systems, such as Ciao and YAP, have been adapted to support bag of tasks style execution so that they can exploit massive parallelism in grid systems [6, 12].

```
top - 16:48:38 up 43 days,  1:41,  1 user,  load average: 4.00, 4.00, 4.00
Tasks: 139 total,   6 running, 133 sleeping,   0 stopped,   0 zombie
Cpu(s): 50.0% us,  0.0% sy,  0.0% ni, 50.0% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:  16409824k total, 10484280k used,  5925544k free,   174136k buffers
Swap:  2040244k total,        0k used,  2040244k free,  7525096k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19577 vitor     25   0  682m 522m 1560 R  100  3.3  19365:53 yap
19581 vitor     25   0  682m 522m 1560 R  100  3.3  19368:01 yap
19580 vitor     25   0  682m 522m 1560 R  100  3.3  19368:44 yap
19582 vitor     25   0  682m 522m 1560 R  100  3.3  19368:24 yap
```

Fig. 1. A Multi-Core in Action

The last few years have seen a major change in this picture. With multi-core processors, *parallelism is just available.* Fig. 1 shows an example of usage of a dual processor machine, with each CPU being four-core, for a total of eight available cores. The machine is also used as a workstation, so only four cores are being used for background tasks [1].

Arguably, such bag-of-tasks parallelism is just *boring*: there is hardly much challenge in launching four processes and waiting for their outcome. But, it is very *useful*, as one is, at least in principle, four times faster (and still has a useful desktop machine), with very little work.

We believe that the challenge facing declarative programming, and we will discuss logic programming here, is to make other forms of parallelism as boring as possible. But to do so, some issues have to be debated first:

– Although there are good reasons to want newer languages, work such as XSB-Prolog and Ciao has shown that incremental progress in current logic programming language design is possible. This makes it the onus of the new language designer(s) to prove that new wine is there, or in other words, that their new approach is widely applicable.
– The last few years have shown the contrast between implicit and explicit parallelism to be largely artificial. Explicit parallelism can be a useful building tool in creating higher-level parallel systems [7]. And implicit parallelism benefits from annotations and other forms of user aid. One can therefore feel rather confident in arguing that there is a continuum of alternatives, and that ideally it should be possible for the programmer to move smoothly in choosing the combination that better suits her or his needs.

[1] unfortunately, this is not the author's machine!

– Work such as KLIC [9] &-Prolog [19], Aurora [25], Muse [1], JAM [11], Andorra-I [41], Penny [26], ACE [29, 28] and the DASWAM [42] addressed research issues and advanced technology, resulting in sophisticated and powerful systems. Which, unfortunately, have proven to be rather hard to maintain. This argues for simpler systems built from reusable-blocks, as in recent work for Ciao [20] and, in a different context, ASP-Prolog [13].
– Logic Programming systems run user tasks, and therefore must interact with the user's environment. This often includes Input/Output and data-base operations, which therefore may be key to efficient execution. It is arguably the case that side-effects have been seen as an obstacle in the race for parallelism and either ignored or set aside [18, 21, 27, 17]. But it does not need to be thus. A good example is data-base support for tabling, which is usually implemented by storing tabled solutions in *tries*. In this case, a parallel implementation can understand the goal of a data-base operation, and exploit parallelism, with excellent results [33]. We believe that it is critical to progress in this direction.
– At the end of the day, it will be the ability to actually run real applications that will decide whether the work will be worthwhile. It has been argued that parallel logic programming had no real applications. This is unfair, as a number of applications have been developed: knowledge-based systems [2], natural language processing [30], multimedia [34], and model checking [32]. More to the point, one can argue that such applications did not include some of the major applications of LP and that they had to compete for scarce parallel resources. As the latter problem transforms from a problem into a motivation, research on the former becomes all the more important.

In a nutshell, we believe that this discussion distills itself into three simple commandments:

– Thou shall use Modular System Construction, so that thou shall be able to maintain and reuse thou code!
– Thou shall Provide High-Level Data Structures, so that the user needs shall be apparent to thou!
– Though shall study and understand Real Applications, so that they shall be the salt of thou work!

Can we apply those commandments in a principled way? We would like to discuss the application of the three commandments but we will focus on the first commandment in this work. We discuss how a specific Prolog system, the YAP Prolog system [37], has been adapted to support parallelism. In a similar fashion to other Prolog systems such as SICStus Prolog [3] and Ciao [20], YAP included some support for shared-memory implicit parallelism, in this case or-parallelism. Building on this support, it was possible to implement a thread library that supports explicit parallelism. On a different vein, the system has also been used for distributed programming and for grid style computation. We compare the costs of the three approaches, and study how our simple commandments can be obeyed, if at all.

2 The YAP Prolog System

The YAP Prolog system was originally developed by Luís Damas and Vítor Santos Costa towards being a high-performance Prolog system [37]. The system includes a number of components, described in Fig. 2. We distinguish three particularly complex components: the abstract emulator (marked as red), the compiler (marked as blue), and the memory management routines, including the garbage collector (marked as gray). Edges show how components depend on each-other.

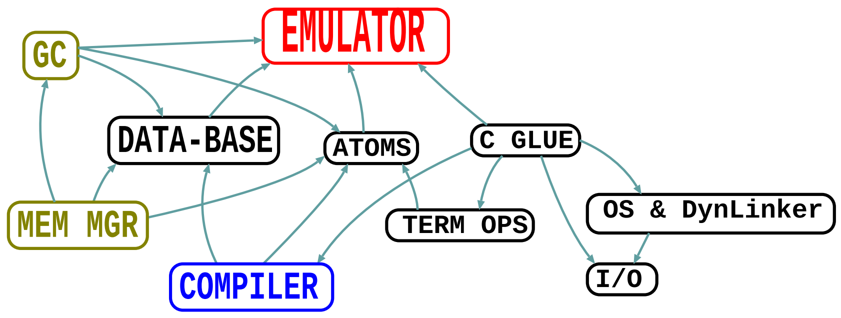

Fig. 2. The Structure of the YAP System

The core component of the system is a WAM abstract machine emulator [36]. Currently, the emulator implements close to 400 different instructions. The emulator interacts directly with three key components:

1. The *C-Glue* provides the interface between the abstract engine and most libraries. It relies on a a set of abstract types, such as Constant, Compound Term, Term, Atom, Functor and a set of constructors and destructors. These primitives are used by all components of the system.
2. The *Atom-Table* is organized as a hash-table and maintains all constants in the system. As in traditional LISP systems [15], it is used as a starting point for all other property lookup.
3. The *Data-Base* supports compiled code, both dynamic and static, and a term database, accessed through the `record` family of built-ins. There are also smaller data-bases: an operator data-base, and a small constant data-base.

These components require dynamically allocated memory:

1. The *Garbage Collector* [8] and the closely-associated *Stack Shifter* can interrupt the engine to compress stacks, clean up dead code, and expand memory regions.

2. The Memory Allocator provides memory allocation and deallocation services for the system. YAP includes three different allocators: **(i)** the original allocator asks the system for a large chunk of memory, allocates big chunks for the stacks, and manages the rest of memory with a greedy algorithm; **(ii)** the default one allocates a huge chunk of memory and expands it dynamically, but uses the Doug Lea allocator to manage memory [23]; **(iii)** last, YAP can just use the standard library routines for memory allocation.
   We have found out that the greedy allocator will not work well on large applications. The Doug Lea allocator has allocator better performance, and is in fact traditionally used by several system libraries.

The full functionality of the system requires extra modules:

1. Input/Output operations are obviously required for the system to be useful.
2. Term operations such as term comparison, are important in actual applications.
3. The Operating System interface allows access to a number of important features. Access to the Dynamic Linker allows run-time extensibility.

Finally, the YAP compiler supports clause-level compilation and dynamic compilation of indices [40].

3 Supporting Parallelism in YAP

Our first commandment says that one should be able to extend the system modularly. In other words, ideally we should implement parallelism by extending the system with a new module, and would not need to rewrite code on the remaining of the system. Unfortunately, and as often is the case, such a commandment may be hard to obey. YAP is an ideal platform to study the problem as it has been used to implement a large number of different approaches. Next, we discuss some of these approaches: we shall start from the methodologies associated with coarser grain-size.

Grid-Support With Condor YAP can be run in grid systems using two approaches: in the common *as is* approach the system is just transfered without changes; otherwise, the system can be adapted to support libraries such as condor [43]. The condor library is particularly interesting because it allows "transparent" check-pointing and migration of jobs, which may become useful as idle cores in networks of workstations become more and more available [12]. Fig. 3 shows where changes were necessary: notice that these changes correspond support the `universe` condor environment on a circa 2004 version of condor.

The changes are required to operate under the more limited functionality available in the condor universe. They essentially drop some of the memory manager functionality, as YAP now **has** to use the standard library; drop dynamic linking as condor requires static linking; and change some time and file access primitives, again due to limitations in the condor programming environment.

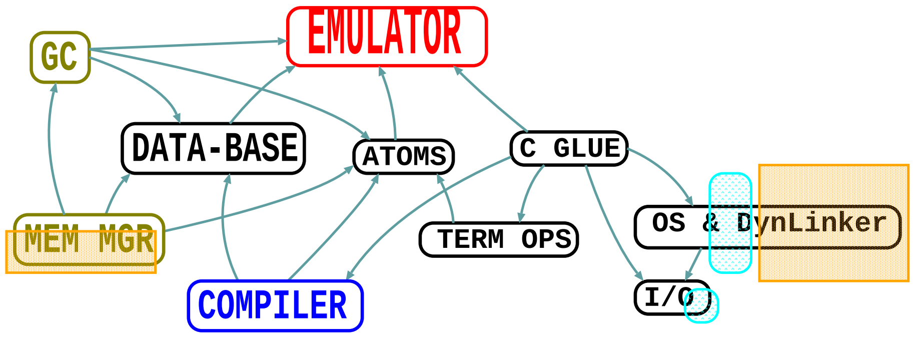

Fig. 3. The Structure of the YAP System with Condor support: square boxes correspond to dropped functionality, round boxes correspond to changes

Arguably, such changes are mostly modular. On the other hand, they are more about *dropping* modules than actually extending a system with new modules. In that sense, they can be seen as a minor, but necessary, sin.

Distributed Processing with MPI The YAP system includes support for two different MPI libraries: MPICH [4] and LAM [5]. The two interfaces were developed independently but operate under similar principles. They provide a low-level interface that allows one to use basic MPI functionality, while exporting and importing Prolog terms as messages. The implementation is shown in Fig. 4.

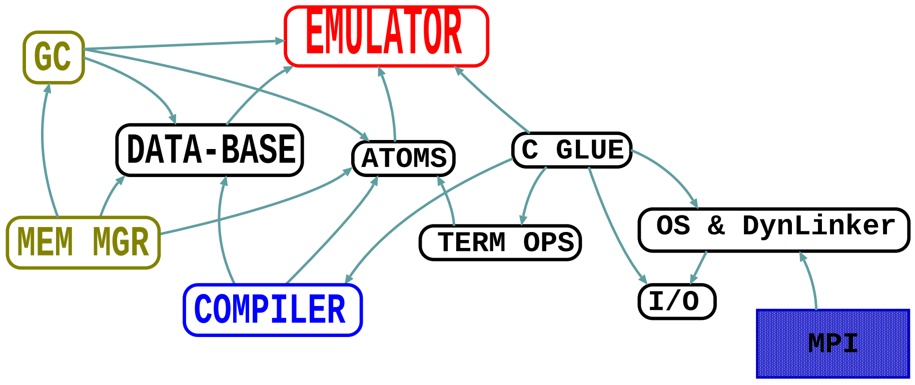

Fig. 4. The Structure of the YAP System with MPI support: the grey square box corresponds to the new functionality

As the picture shows, both interfaces operate as a new module that links to the system through the `C`-interface. The interfaces did not require changes to Prolog, although they would benefit from functionality in the term libraries. Arguably, such an implementation is not the most efficient, but it is the easier to update and maintain, and follows perfectly our first commandment.

Threads The YAP system includes support for Posix p-threads in the style of the SWI-Prolog thread library [45]. In this approach, threads run on separate stacks but share access to the data-base. The implementation is presented in Fig. 5.

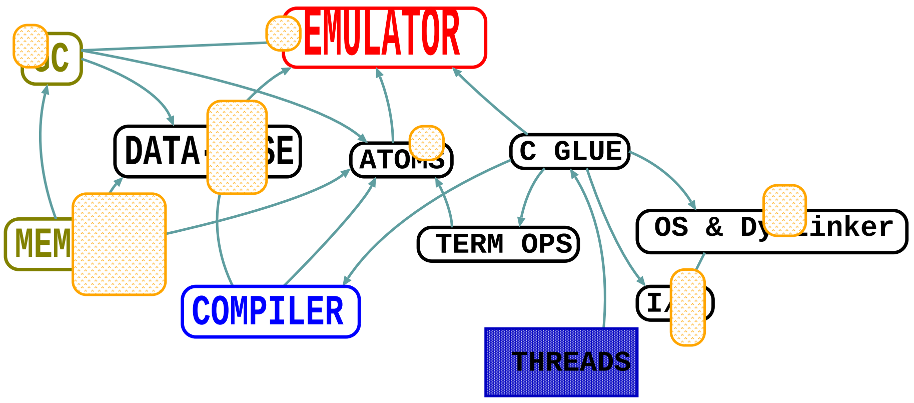

Fig. 5. The Structure of the YAP System with thread support: the grey square box corresponds to the new functionality, the round corner boxes correspond to significant changes in the pre-existing code

As the picture shows, supporting threads requires major changes throughout the system. Most of these changes have to do with the need to synchronize access to the data-base and the Input/Output. In some more detail:

1. One needs a new module for threading. Notice that this module is now plugged in more closely to the system, as communication is expected to be quite more intensive.
2. One needs major changes to the Data-Base, due to the need to synchronize access to dynamic structures, such as the index trees [40] and dynamic predicates. Such data-structures are quite important in large programs. These changes then ripple down to the garbage collector and to the emulator.
3. The memory allocator needs to support concurrency and multiple stacks. In fact, the easiest solution is to rely on the system library, as for condor.
4. The Atom-Table needs to support concurrent access.

Most of the complexity stems from the concurrent accesses to the data-base. The changes are quite intrusive and hard to debug, as it is often the case with concurrent systems. Memory management is a second problem: threads require extra memory to support locks, that depending on the grain size, may be quite frequent. Note that several approaches are possible, ranging from a big central lock to fine–grained access with specialized data-structures, such as read-write locks. Supporting threads is therefore not modular, but we have been able to provide most of the functionality in the non-threaded system (except for the atom garbage collector). As threads may be used to build other primitives, they may arguably be a necessary sin.

Native Or-Parallelism YAP includes code for a native implementation of three different models of or-parallelism: COWL, stack-copying, and the Sparse Binding Arrays [39]. The implementation is further complicated by the need to support tabling [32]. Although the implementation is not currently being actively developed, it is still in the code. The implementation is presented in Fig. 3.

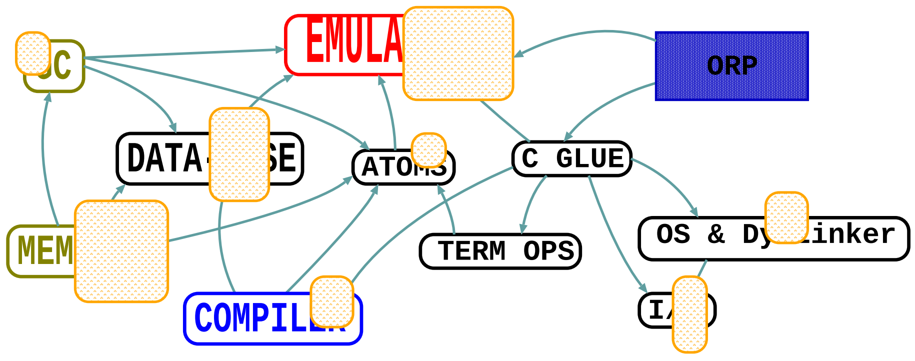

Fig. 6. The Structure of the YAP System with native or-parallelism: the grey square box corresponds to the new functionality, the round corner boxes correspond to significant changes in the pre-existing code

It should be clear that there is extensive overlap between native or-parallelism and threads: this is because both implementations can perform concurrent access to the data-base and concurrent I/O, although concurrent access in or-parallel systems may be unnecessary if one requires a sequential order [18, 17]. Or-parallel requires major changes to:

1. The memory allocator needs several low-level optimizations to support or-parallelism, namely for stack-copying.
2. The emulator needs several new instructions to run or-parallelism. Furthermore, some useful invariants in the sequential system will now be broken. The trailing mechanism is particularly affected.

The actual impact of these changes depends on the particular model, with the process-based model of the COWL being the least intrusive [35] and the shared-memory approach of the SBA being the most intrusive [10], as it requires to share the stacks and to create private images. All the models do require new instructions and major changes to key data-structures, such as choice-points.

3.1 Evaluation

Table 1 shows how pervasive the changes are in YAP's `C`-code. We use Nuno Fonseca's LAM interface in this study. The first column shows the number of `define` directives needed, and the second column shows how many files were affected (out of a total of 104). The third column shows how much new code was needed for the new functionality. Up to a third of all system files (including most header files) need to be changed in some way to support threads or implicit parallelism. The changes to support *condor* are much less extensive, and they mostly have to with the need to support a different memory allocator. As expected, although or-parallelism requires more changes than threads, the changes are largely on the same files. In fact, around 141 defines are shared between or-parallelism and threads.

	#`define` needed	Files Affected	New Lines
`CONDOR`	3 (47)	3	0
`MPI`	0	0	1652
! `THREADS`	231 (47)	33	1280
`ORP`	680	33	4222

Table 1. Number of `define` used to separate parallelism specific code, number of files where such defines were used and extra code needed (including comments). Changes required to use the system allocator are in brackets.

It is unsurprising that implementing parallelism requires more extra code. Of this code, about 300 lines of code implement a locking library which is shared with the thread implementation. So the actual cost of writing the thread library is under a thousand lines of code, mostly on packing and unpacking arguments when calling `p-thread` functions. This is smaller than the MPI interface, as the latter has to copy and receive terms from messages.

Is there a runtime cost for this functionality? Previous reports indicate costs on the order of 5% for applications with SWI-Prolog [45], and with or-parallelism in YAP [32]. An interesting worst possible situation is the case where almost every operation needs to be protected by a lock. In order to study this setting, we compare YAP without and with thread support on small, data-base intensive examples. Our comparison was run on a dual-CPU machine, where each CPU is a 4 core Intel(R) Xeon(R) CPU E5345 running at 2.33GHz. The machine has 16GB memory, and runs RedHat Entreprise Linux release 4 with kernel 2.6.9. We use Yap-5.1.2 compiled for the `x86_64` architecture. The machine was connected to the network, and the file-system was NFS; the experiments were performed through `ssh` access, the machine was otherwise idle. The tests are as follows: **(i)** `t1` accesses a dynamic predicate with a single fact; this requires holding a lock: **(ii)** `t2` asserts and retracts a dynamic predicate with a single fact; **(iii)** `t3` asserts a fact of the form `t3(RandomNumber)`; **(iv)** `t4` retracts a fact of the form `t4(RandomNumber)`; **(v)** `t5` asserts a fact where the argument is a list of length up to 10 and branching factor up to 500; **(vi)** `t6` asserts a fact where the argument is a list of length up to 10 and branching factor up to 2 (hence there will be more repeated facts).

Results are shown in Table 2. We compare two versions with threads: in the first version, threads are implemented using `C`-code from the Linux kernel that uses hardware instruction. In the second version, we call the `P-Thread` routines. The results show that there is indeed an overhead, that the overhead can be very large if the application just updates the data-base, but that it tends to reduce as operation cost increases. The results are also somewhat disappointing in that we would expect locking on the `P-Thread` library to have substantially improved in newer thread libraries. This does not seem to be the case: locking performance is still very much under par.

	`t1`	`t2`	`t3`	`t4`	`t5`	`t6`
`NO-THREADS`	11	642	298	792	1354	1379
`THREADS-USER LOCKING`	11	1144	480	1558	1520	1527
`THREADS-PTHREAD LOCKING`	25	1717	767	1851	2116	2163

Table 2. Running time for 500000 iterations of simple data-base access predicates, using a native implementation, user code for locking, and the p-thread library locking routines.

The results also show that locking can indeed decrease system performance. The data-base is controlled by reader-writer locks, which are called at 263 points in the code. Standard locks are called at 113 points in the code. Locking is required whenever accessing a dynamic predicate. YAP does not need to lock the indexing code for static procedures because it is write once [40].

3.2 Parallel Execution

The usefulness of the techniques discussed here clearly depends on how much the underlying computer architecture can support them. To validate whether it is worthwhile to exploit these machines, we experimented a number of simple benchmarks on the multi-core machine. The 3 experiments include just accessing a static and a dynamic fact, building a long list, and randomly accessing a very large compound term through `arg/3`. Experiments are run in by *executing the same code at N threads*: the only communication is when accessing the read-lock that protects the dynamic predicate in the second experiment. All other experiments have *no* synchronization within YAP. The ideal result would be constant-time, and except for the second benchmark, slowdowns should be caused by limitations in the multi-core architecture.

Cores	1	2	3	4	5	6	7	8
`a(1)`	1771	1780	1795	1794	1772	1793	1798	1833
`a(1)` (dynamic)	1635	1623	1636	1629	1617	1625	1623	1684
`mklist(10000,_)`	435	438	437	436	443	443	450	491
`mklist(100000,_)`	4540	4628	4660	4438	4737	5071	5970	6658
`mklist(1000000,_)`	473232	47605	47692	46674	50014	55351	62401	70228
`arg(Rand,10000,_)`	693	1523	6914	12885	13480	14469	14297	15235
`arg(Rand,100000,_)`	866	2765	7713	14108	15420	16646	16137	17245
`arg(Rand,1000000,_)`	1075	3203	7869	14688	16302	16978	17275	18469

Table 3. Running time in msec for 20,000,000 iterations of a simple query, for 200 iterations of the `mklist/2` predicate, and for 2,000,000 random accesses to a large term. Every thread repeats the original task, hence ideal performance would be constant time.

The results show a complex story. First, they show almost perfect parallelism for the simple query: in this case, the cores can happily process away in their local caches, and performance is excellent even when all cores are in use. Second, they show that a limited amount of synchronization has no impact on system performance: the static and dynamic versions of the code execute in much the same way.

The results for `mklist` show excellent performance for the two list smaller lists, with 160KB and 1600KB. Performance drops somewhat for larger number of cores when we construct the 16MB list. In this case, we have a slowdown of 60% when the 8 cores construct the 8MB lists in parallel.

The `arg/3` results were the most surprising. The simple benchmark was chosen as an example of stressing the cache system. It does its task well. With two cores, it is just faster to run the benchmarks sequentially than to run them in the separate cores. The picture grows worse for 3 cores and for 4 cores: in fact, adding the 4th core consistently halves performance! It is also interesting that after 4 cores performance degrades more gracefully (maybe because contention is now bad enough). The size of the term is not particularly important: the effects are very clear with a 800KB terms, and there is only a small price to access an 80MB term.

4 Future Work

We started this work assuming that multi-cores will make parallel programming boring, and hoping that parallel logic programming would follow. We will have to wait: multi-core architectures are complex, with memory performance far more critical than for traditional shared memory machines. Programming these machines may require understanding memory access patterns, admittedly a harder task in the context of declarative languages.

This leads us to commandment number two: providing high-level data structures that are well understood and that can be profiled and analised with confidence may be what makes parallel logic programming successful. And this in turn leads to commandment number three, and to the question we should have started from: what do logic programmers need?

There is not a single answer to this question. Our experience shows that often people just want to run similar tasks. In this case, the question is "how does Prolog access memory?", and this is an hard problem by itself [24]. But, one can go one step further by looking at actual applications. In the author's case, at the time of writing this paper, he was interested in two main applications in the area of Statistical Relational Learning.

The Problog language combines logic with probabilities by saying that a clause may be true [31]. The probability of a goal is evaluated by combining the probabilities of all paths that prove the goal. This is very close to traditional or-parallelism, except that successful (and interrupted) proofs must be stored away. Doing this sequentially would kill parallelism, But there is no real reason to do so. Solving this problem is thus a question of designing a data-structure that can store proofs and that allows concurrent updates, such as, say, a trie [33]!

The CLP($\mathcal{BN}$) language can be used to specify graphical models, such as Hidden Markov Models [38]. One interesting query in these models is to find the most likely explanation to a sequence of observations. The Viterbi algorithm is the main tool for this task. The algorithm implements dynamic programming and proceeds in two steps. Computation is dominated by the forward step where it steps across all nodes in the graph following a dominance order. The first CLP($\mathcal{BN}$) implementation used constraints to represent the node, topologically sorted them, and then run the algorithm. This was elegant, but expensive. A recent implementation does not generate the nodes. Instead, it generates a set of instructions describing the graph and applies them to every element in the sequence. Even so, the application can take seconds for larger networks. The application is an example of data-flow parallelism, but we have observed that for large networks chunks tend to be somewhat independent, so sub-computation can proceed in parallel as long as we have shared access to the state. Can we exploit independent and-parallelism in this context?

It would be a mistake to ignore the huge amount of work in parallel logic programming: much progress was made, and many lessons were learned. We understand the main issues in implicit parallelism, and we know where the main pitfalls wait for us. We should thus start from the lessons we learned. And we should build something new, and better.

Acknowledgments

The author gratefully acknowledges the support of the project STAMPA, FCT Project PTDC/EIA/67738/2006. The work has been partially supported by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC. The work was motivated by discussions at the CICLOPS07 pannel: the author thanks B. Demoen, D. S. Warren, E. Pontelli, F. Silva, G. Gupta, I. Dutra, M. Hermenegildo, R. Rocha, and S. Abreu for sharing their ideas.

References

1. K. A. M. Ali and R. Karlsson. The Muse approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162 (or 129–160??), Apr. 1990.
2. K. A. M. Ali and R. Karlsson. OR-Parallel Speedups in a Knowledge Based System: on Muse and Aurora. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 739–745, ICOT, Japan, 1992. Association for Computing Machinery.
3. J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjoland, and J. Widén. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, November 1997. SICS Technical Report T93-01.
4. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
5. G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
6. M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.
7. A. Casas, M. Carro, and M. Hermenegildo. Towards High-Level Execution Primitives for And-parallelism: Preliminary Results. In *CICLOPS 2007: 7^{th} Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 102–116, Porto, 2007.
8. L. F. Castro and V. Santos Costa. Understanding Memory Management in Prolog Systems. In *Proceedings of Logic Programming, 17th International Conference, ICLP 2001*, volume 2237 of *Lecture Notes in Computer Science*, pages 11–26, Paphos, Cyprus, November 2001.
9. T. Chikayama, T. Fujise, and H. Yashiro. A portable and reasonably efficient implementation of KL1. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, page 833, Budapest, Hungary, 1993. The MIT Press.
10. M. E. Correia, F. Silva, and V. Santos Costa. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *Proceedings of the 1997 International Logic Programming Symposium*, pages 117–131. MIT Press, October 1997. Also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.
11. J. A. Crammond. The abstract machine and implementation of parallel parlog. *New Generation Computing*, 10(4):385–422, 1992.
12. I. C. Dutra, D. Page, V. Santos Costa, J. W. Shavlik, and M. Waddell. Towards automatic management of embarassingly parallel applications. In *Proceedings of*

Europar 2003, volume 2790 of *Lecture Notes in Computer Science*, pages 509–516, Klagenfurt, Austria, August 2003. Springer Verlag.

13. O. El-Khatib, E. Pontelli, and T. C. Son. Integrating an answer set solver into prolog: Asp-prolog. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 399–404. Springer, 2005.
14. N. A. Fonseca, F. Silva, and R. Camacho. April - An Inductive Logic Programming System. In F. M, V. W, K. B, and L. A, editors, *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA06)*, volume 4160 of *Lecture Notes in Artificial Intelligence*, pages 481–484, Liverpool, September 2006. Springer-Verlag.
15. R. P. Gabriel. *Performance and evaluation of Lisp systems*. MIT Press, 1985.
16. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):1–131, 2001.
17. G. Gupta and V. Santos Costa. Cuts and Side-Effects in And-Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, April 1996.
18. B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1988*, pages 831–840. ICOT, 1988.
19. M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
20. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.
21. L. V. Kalé, D. A. Padua, and D. C. Sehr. OR-Parallel execution of Prolog with side effects. *The Journal of Supercomputing*, 1988.
22. S. T. Konstantopoulos. A data-parallel version of Aleph. In R. Camacho and A. Srinivasan, editors, *Proc. of the Workshop on Parallel and Distributed Computing for Machine Learning, ECML/PKDD 2003*, 2003.
23. D. Lea. *A Memory Allocator*.
24. R. Lopes, L. F. Castro, and V. Santos Costa. From Simulation to Practice: Cache Performance Study of a Prolog System. In *ACM SIGPLAN Workshop on Memory System Performance, Berlin, Germany*, June 2002. SIGPLAN Notices vol 38(2), February 2003, pages 56–64.
25. E. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.
26. J. Montelius and K. A. M. Ali. An And/Or-Parallel Implementation of AKL. *New Generation Computing*, 14(1), 1996.
27. K. Muthukumar and M. V. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 80–97. MIT Press, June 1989.
28. E. Pontelli and G. Gupta. Data and-parallel logic programming in &ace. In *7th IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1995.

29. E. Pontelli, G. Gupta, and M. V. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium.* IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.
30. E. Pontelli, G. Gupta, J. Wiebe, and D. Farwell. Natural language multiprocessing: A case study. In *AAAI/IAAI*, pages 76–82, 1998.
31. L. D. Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In M. M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.
32. R. Rocha, F. Silva, and V. S. Costa. On Applying Or-Parallelism and Tabling to Logic Programs. *Theory and Practice of Logic Programming Systems*, 5(1-2):161–205, 2005.
33. R. Rocha, F. Silva, and V. Santos Costa. Achieving Scalability in Parallel Tabled Logic Programs. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPPDPS02), Fort Lauderdale, Florida, USA*, April 2002.
34. S. W. Ryan and A. K. Bansal. A scalable distributed multimedia knowledge retrieval system on a cluster of heterogeneous high performance architectures. *International Journal on Artificial Intelligence Tools*, 9(3):343–367, 2000.
35. V. Santos Costa. Cowl: Copy-on-write for logic programs. In *Proceedings of the IPPS/SPDP99*, pages 720–727. IEEE Computer Press, May 1999.
36. V. Santos Costa. Optimising bytecode emulation for prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.
37. V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*, 2002. http://www.ncc.up.pt/~vsc/Yap.
38. V. Santos Costa, C. D. Page, and J. Cussens. *Probabilistic Inductive Logic Programming*, chapter CLP($\mathcal{BN}$): Constraint Logic Programming for Probabilisti c Knowledge. Springer-Verlag, 2007. (to appear).
39. V. Santos Costa, R. Rocha, and F. Silva. Novel Models for Or-Parallel Logic Programs: A Performance Analysis. In *Proceedings of EuroPar2000, LNCS 1900*, pages 744–753, September 2000.
40. V. Santos Costa, K. Sagonas, and R. Lopes. Demand-driven indexing of prolog clauses. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 305–409. Springer, 2007.
41. 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.
42. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3), 1996.
43. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.
44. D. H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool.* PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.
45. J. Wielemaker. Native preemptive threads in swi-prolog. In C. Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2003.