YapOr: an Or-Parallel Prolog System based on Environment Copying

Ricardo Rocha   Fernando Silva   Vítor Santos Costa

Technical Report Series: DCC-97-14

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

`http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html`

YapOr: an Or-Parallel Prolog System based on Environment Copying

Ricardo Rocha Fernando Silva Vítor Santos Costa

DCC-FC & LIACC, University of Porto,
R. do Campo Alegre, 823 4150 Porto, Portugal
{ricroc,fds,vsc}@ncc.up.pt

Abstract. Prolog compilers have proved to be highly efficient for standard sequential architectures. The efficiency of these implementations and the declarativeness of the Prolog language have kindled interest on implementations for parallel architectures. YapOr is an Or-parallel system that extends Yap's sequential execution model to exploit implicit or-parallelism in Prolog programs. YapOr is based on the environment copy model, as first implemented in Muse.
To develop YapOr, it was necessary to solve some important issues, such as the design of data structures to support parallel processing, the implementation of the incremental copying technique, the development of a memory organization able to answer with efficiency to parallel processing and to the incremental copying in particular, the implementation of the scheduling strategies, the design of an interface between the scheduler and the engine, the implementation of the sharing work process, and support for the cut builtin.
An initial evaluation of YapOr performance showed that it achieves very good performance on a large set of benchmark programs. Indeed, YapOr compares favorably with a mature parallel Prolog system such as Muse, both in terms of base speed and in terms of speedups.
Keywords: Parallel Logic Programming, Scheduling, Performance Evaluation.

1 Introduction

Prolog is arguably the most important logic programming language. It has been used for all kinds of symbolic applications, ranging from Artificial Intelligence to Database or Network Management. Traditional implementations of Prolog were designed for the common, general-purpose sequential computers. In fact, WAM [War83] based Prolog compilers proved to be highly efficient for standard sequential architectures and have helped to make Prolog a popular programming language. The efficiency of sequential Prolog implementations and the declarativeness of the language have kindled interest on implementation for parallel architectures. In these systems, several processors work together to speedup the execution of a program [GACH96]. Parallel implementations of Prolog should obtain better performance for current programs, whilst expanding the range of applications we can solve with this language.

Two main forms of implicit parallelism are present in logic programs [GACH96]. *And-parallelism* corresponds to the parallel evaluation of the various goals in the body of a clause. This form of parallelism is usually further subdivided into *Independent And-parallelism* in which the goals are independent (don't share variables), and *Dependent And-parallelism* in which some goals share some variables with others. In contrast, *Or-parallelism* corresponds to the parallel execution of alternative clauses for a given predicate goal. This latter form of parallelism is, in principle simpler to exploit since it is known that the alternative branches in the search tree are independent of each other, hence imposing less synchronization between the processors (or workers) that are executing in parallel.

Original research on the area resulted in several systems that successfully supported either and-parallelism or or-parallelism. These systems were shown to obtain good performance for classical shared-memory parallel machines, such as the Sequent Symmetry. Towards more flexible execution, recent research in the area has investigated how to combine both And- and Or-parallelism [CSC97], and how to support extensions to logic programming such as constraints and tabling [RSC97].

Of the forms of parallelism available in logic programs, or-parallelism is arguably one of the most successful. Experience has shown that or-parallel systems can obtain very good speedups for a large range of applications, such as applications that require search. Designers of or-parallel systems must address two main problems, namely scheduling and variable binding representation. Work in or-parallel systems arises irregularly, and requires careful scheduling. Several strategies have been proposed to this problem [CS89, AK90a, BRS91, Sin91, Sil93].

The binding representation problem is a fundamental problem that arises because the same variable may receive different bindings in different or-branches. A number of approaches [GJ93] have been presented to tackle the problem. Two successful ones are environment copying, as used in Muse [AK90b], and binding arrays, as used in Aurora [LBD+88]. In the copying approach, each worker maintains its own copy of the path in the search tree it is exploring. Whenever work needs to be shared, the worker that is moving down the tree copies the stacks from the worker that is giving the work. In this approach, data sharing between workers only happens through an auxiliary data structure associated with choice-points.

In contrast, in the binding array approach work stacks are shared. To obtain efficient access, each worker maintains a private data structure, the binding array, where it stores its conditional bindings. To allow for quick access to the binding of a variable the binding array is implemented as an array, indexed by the number of variables that have been created in the current branch. The same number is also stored in the variable itself, thus giving constant-time access to private variable bindings.

Initial implementations of Or-parallelism, such as Aurora or Muse, relied on detailed knowledge of a specific Prolog system, SICStus Prolog. Further, they were designed for the original shared memory machines, such as the Sequent Symmetry. Modern Prolog systems, even if emulator based, have made substantial improvements in sequential performance. These improvements largely result from the fact that though most Prolog systems are still based on the Warren Abstract Machine, they have a rather different internal structure than the original SICStus Prolog. Moreover, the impressive improvements on CPU performance over the last years have not been followed by corresponding bus and memory performance. As a result, modern parallel machines show a much higher latency, as measured by the number of CPU clock cycles, than original parallel architectures.

The question therefore arises of whether the good results previously obtained with Muse or Aurora in Sequent style machines are repeatable with other Prolog systems in modern parallel machines. In this work, we present YapOr, an Or-Parallel Prolog system, that is based on the high performance Yap Prolog compiler [DSCRA89], and demonstrate that the low overheads and good parallel speedups are in fact repeatable for a new system in a very different architecture.

The implementation of Or-Parallelism in YapOr is largely based on the environment copy model as first introduced by Ali and Karlson in the Muse system [AK90b, AK90a, Kar92]. We chose the environment copying model because of its simplicity, that made it easier to adapt to a complex Prolog system such as Yap, and because of its efficiency, as Muse had consistently demonstrated less overheads than competing Or-Parallel systems such as Aurora. Note that there is some interest in supporting other or-parallel models, and we designed the system in such a way as to be easily adaptable to alternative execution models.

The elegance of the original Muse design was one extra reason we chose to follow this design. Still, and as explained before, the substantial differences between YapOr and Muse resulted in several contributions from our design, such as:

- YapOr uses novel memory organization and locking mechanisms to ensure mutual exclusion.
- We present different handling of the backtracking mechanism, as an extension of WAM instructions, and not through a SICStus specific mechanism.
- As in the original Yap, we use just one stack for environments and choice-points, in opposition to SICStus which uses two. This requires adjustments to the sharing and synchronization procedures when two workers share work. This also requires for different formulae to be used when calculating the portions of stacks that have to be copied when sharing work takes place.

- YapOr introduces a new protocol between workers to handle the cut predicate.
- We introduce new scheme to support the solutions that are being found by the system and that may correspond to speculative work.

A first YapOr implementation has integrated in the freely distributable YAP system. Performance analysis showed that sequential performance was superior to that of the original Muse, and better than the latest Muse as available with the current commercial implementation of SICStus Prolog.

The remainder of the paper is organized as follows. First we present the general concepts of the Environment Copying Model. We next introduce the major implementation issues in YapOr. We then give a detailed performance analysis for a standard set of benchmarks. Last, we present our conclusions and discuss some further work.

2 The Environment Copying Model

As previous systems, YapOr uses the multi-sequential approach [LBD+88]. In this system Prolog *workers* (or engines, or processors or processes) are expected to spend most of their time performing reductions, corresponding to useful work. When they have no more goals or branches to try, workers search for work from fellow workers. Which workers they ask for work from and which work they receive is a function of the *scheduler* code.

Basic Execution Model

Parallel execution of a program is performed by a set of workers. Initially all workers but one are *idle*, that is, looking for their first work assignment. Only one worker, say `P`, starts executing the initial query as a normal Prolog engine. Whenever `P` executes a predicate that matches several execution alternatives, it creates a choice-point (or node) in its local stack to save the state of the computation at predicate entry. This choice-point marks the presence of potential work to be performed in parallel.

As soon as an idle worker find that there is work in the system the worker will request work directly from a busy worker. Consider, for example, that worker `Q` requests work from worker `P`. If `P` has work to give, it will have to share its local points with `Q`. To do so, worker `P` must turn its choice-points *public* first. In an environment copying based model this operation is implemented by creating or-frames in a shared space that allow access to the newly shared choice-point. Next, worker `P` will hand `Q` a pointer to the last alternative it made sharable, or in other words, to the bottom-most or-frame still with unexplored alternatives.

The next step is taken by worker `Q`. In order for worker `Q` to take a new task it must copy the computation state for worker `P` up to the node from which the new task is to be taken. After copying, worker `Q` must synchronize its status with the newly copied choice-point. This is implemented by first simulating a failure to the choice-point and then by backtracking to the next available alternative within that node and starting its execution as a normal sequential Prolog engine would.

At some point, a worker will fully explore its current sub-tree and become idle again. In this case it will go again into the scheduling state and start looking for busy workers in order to request work from them. It thus enters the behavior just described for `Q`.

Eventually the execution tree will be fully explored and execution will terminate with all workers idle.

Incremental copying

The work sharing operation poses a major overhead to the system as it involves the copying of the executions stacks between workers. Hence, an incremental copying strategy has been devised in order to keep this source of overhead to a minimum.

The main goal of this strategy is to position the workers involved in the operation in the same node of the search tree, leaving them with the same computational state. To achieve this goal, incremental copying makes the receiving worker keep the part of its state that is consistent with the giving worker, and only copy the differences between both.

The implementation of this strategy can be better understood through figure 1. Suppose worker `Q` does not find available work in its sub-tree, and that there is a worker `P` with work to share. Consider that `Q` asks `P` for work, and that `Q` backtracks to the first node that is common to `P`, therefore becoming partially consistent with part of `P`. Now worker `P` decides to share its private nodes by placing them in the shared address space, followed by a copy of the differences between `P` and `Q`. These are calculated through the information stored in the common node found by `Q` and in the segments at the top of the local, heap and trail stacks of `P`.

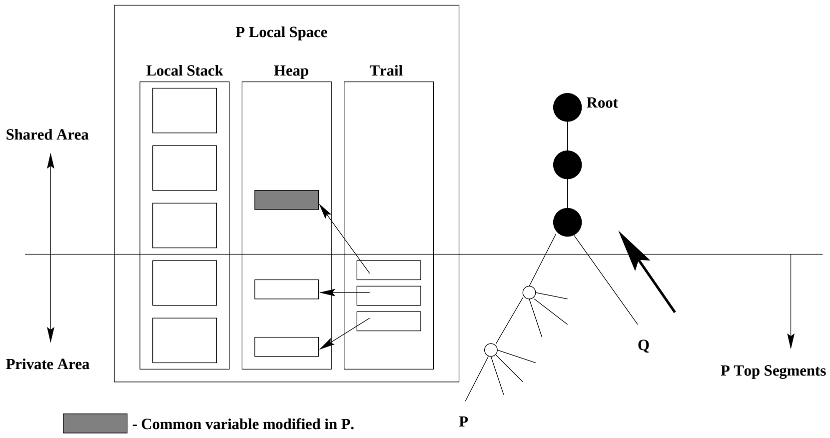

Fig. 1. Some aspects of incremental copying.

Work scheduling

We can divide the execution time of a worker in two modes: scheduling mode and engine mode. A worker enters in scheduling mode whenever it runs out of work and starts searching for available work in the shared part of the tree. As soon as it gets a new piece of work, it enters in engine mode. In this mode, a worker runs like a basic Prolog engine.

The scheduler is the system component that is responsible for distributing the available work between the various workers. The scheduler must arrange the workers in the search tree in such a way as that the total time of a parallel execution will be the least possible. The scheduler must also maintain the correctness of Prolog sequential semantics and minimize the scheduling overheads present in operations such as sharing nodes, copying parts of the stacks, backtracking, restoring and undoing previous variable bindings.

To achieve these goals, the scheduler follows the following strategies:

- When a busy worker shares work, it must share all the private nodes it has in that moment, making them public. This will maximize the amount of shared work at any sharing operation and possibly avoid that the requesting worker runs out of work too soon.
- The strategy adopted by the scheduler for work sharing is to select the busy worker such that it holds the highest work load and that simultaneously is nearest to the idle worker. The *work load* is a measure of the amount of unexplored private alternatives, given as number of nodes or of alternatives. *Being near* corresponds to the closest position in the search tree. This strategy maximizes the amount of shared work and minimizes the stacks parts to be copied.

- In order for a sharing operation between two workers to be performed correctly, it is necessary that the idle worker is positioned in a sub-branch of the branch on which the busy worker is working. To guarantee this condition, the idle worker, before requesting work, backtracks to position itself at the first node in the search tree common between itself and the busy worker. This is implemented before the sharing operation so that the busy worker will spend the least time possible in this operation.
- When the scheduler does not find any available work in the system, it must still distribute the idle workers in the search tree in a manner that minimizes the overheads of future sharing operations.

Thus, we can resume the scheduler algorithm to the following idea: when a worker runs out of work, it searches for the nearest unexplored alternative in its branch of the search tree. If there is no such unexplored alternative, it selects a busy worker to share work according to the strategies above. If there is no such a worker, the idle worker tries to move to a better position in the search tree.

There are two alternatives to search for busy workers with excess of work load in the search tree: search below or search above the current node. Idle workers always start to search below the current node. Only if they do not find any busy worker they try searching above. The advantages of selecting a busy worker below instead of above are mainly two. The first is that the idle worker can request immediately the sharing operation, because its current node is already common to the busy worker. It does not need to backtrack in the tree and undo variable bindings. The second advantage is that the idle worker will maintain its relative position in the search tree, but restarting the execution in a bottom level. This corresponds to the bottom-up scheduling strategy of the environment copy model, that has proved to be more efficient than the top-down one.

We said when the scheduler does not find any unexplored alternative in the idle worker branch and finds no busy worker with excess load below or above the idle worker's current node, it tries to move the idle worker to a better position in the search tree. An idle worker moves to a better position, when one of the following situations holds:

1. All the workers below the current node are idle.
2. There are busy workers above and no idle workers upper in its branch.

In the first situation, the idle worker backtracks until it reaches a node where there is at least one busy worker below. In the second one, it backtracks until it reaches the node that contains all the busy workers below. With this scheme, the scheduler tries to distribute the idle workers in such a way as that the probability of finding, as soon as possible, busy workers with excess of work below the corresponding idle workers' current nodes is substantially increased.

3 Extending Yap to support YapOr

We next discuss the main issues in extending the Yap Prolog system to support environment based copying.

Memory Organization

Following the original WAM definition [War83], the Yap Prolog system includes four main memory areas: code area, heap, local stack and trail. The local stack contains both environment frames and choice points. Yap also includes an auxiliary area used to support some internal operations.

The YapOr memory is divided into two big addressing spaces: the global space and a collection of local spaces (see figure 2).

– The **global space** is shared by all workers and contains the data structures necessary to support parallelism. This space is divided in two different areas. The first includes the code area, inherited from Yap, and the global information area to support workers' synchronization. The second is the shared frames area in which three types of parallel frames are allocated during execution: or-frames, answer-frames, and solution frames.

– The **local space** represents one system worker. It contains a local information area with individual information about the worker, and the WAM execution stacks inherited from Yap: heap, local, trail and auxiliary stack.

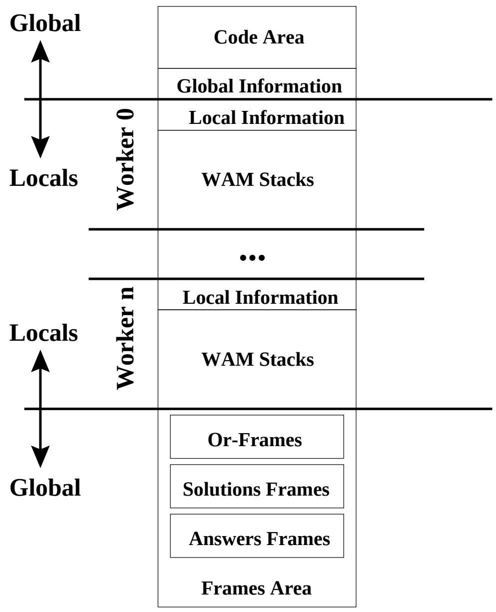

Fig. 2. Memory organization in YapOr.

In order to meet the requirements for efficiency with the incremental copy technique, the memory organization in YapOr has been implemented with the *mmap* function. This function let us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read.

The mapping of all the memory needed to implement the above memory organization follows the principles used in Muse. The starting worker, that is `worker 0`, uses `mmap` to ask for shared memory in the system's parallel initialization phase. Following this mapping the remaining system workers are created, through the use of `fork` function. These workers inherit the memory addressing space previously mapped. Then, each new worker remaps all the local spaces, in such a way that from an individual point of view, its local space starts at the same address. This is done, however, without any change in each relative worker area in the mapped file. This is possible, if each new worker only remaps different memory addresses in the previous areas defined by `worker 0` in the mapped file.

This memory mapping scheme allows for efficient copy operations between workers. The copying is implemented by simply copying directly from one given address to the relative address in the other worker's address space. Reallocation of the address values is not necessary because all the WAM stacks are located at the same fixed address in each worker.

Choice Points and Or-Frames

In order to execute efficiently and in parallel the alternatives of a choice point, avoiding duplication, it is necessary for the choice point to be shared among the workers executing in parallel. Shared choice points reside in frames located in the global shared frames area, therefore making it possible for different workers to reference the choice point. Figure 3 represents the new structure of the choice points. The first six fields are inherited from Yap, while the last two were introduced in YapOr. The `CP_OR-FR` field saves a pointer to the corresponding or-frame associated with the choice point, while the `CP_PUA` field contains the number of private unexplored alternatives in upper choice points, and is used to compute the worker's load.

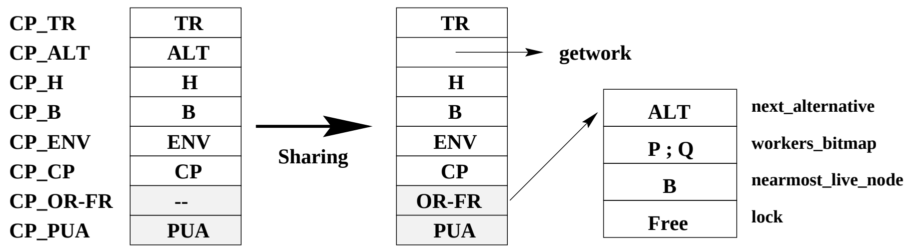

Fig. 3. Sharing a choice point.

One fundamental sharing operation is to make public the private choice points. Figure 3 illustrates the relation between the choice points before and after that operation, and the resulting connection with the correspondent or-frame meanwhile created. The `CP_ALT` and `CP_OR-FR` choice point fields are updated to point respectively to the `getwork` pseudo-instruction (see next section) and to the created or-frame. The or-frame is initialized as follows: the `next_alternative` field stays with the `ALT` pointer, which was previously in the `CP_ALT` choice point field (i.e., the control of the untried alternatives goes into the or-frame); the workers involved in the sharing operation are marked as bits in the `workers_bitmap` field; the `nearmost_live_node` field points to the parent choice point; and the `lock` field stays free to allow the access to the structure.

New Pseudo-Instructions

YapOr introduced only two new instructions. One of them, is the already mentioned `getwork`, and the other is the `getwork_first_time` instruction. These are never generated in the compiler process. They are introduced according to the progress of the parallel execution.

As mentioned earlier, a pointer to special code containing the `getwork` instruction is introduced in `CP_ALT` choice point field when the choice point is being turned public. This way, when a worker backtracks to a shared choice point, it takes the `getwork` instruction in place of the next untried alternative. The execution of this instruction allows for a synchronized access to untried alternatives among the workers sharing the correspondent or-frame.

Whenever the search tree for the top level goal is fully explored, all the workers, except `worker 0`, execute the `getwork_first_time` instruction. This instruction puts the workers in a delay state, waiting for a signal from `worker 0`, indicating the beginning of a new goal. On the other hand, `worker 0` is responsible to give all the solutions encountered for the last exploited goal and to administrate the interface with the user until he makes a new query goal.

Worker's Load

Each worker holds a local register, `load`, as a measure of the number of private unexploited alternatives. The value in this register is used by the scheduler to select the best worker to share work with an idle worker. When a choice point is created, the field `CP_PUA` is initialized with the value of the local untried alternatives in the previous choice points. We do not include in this calculation values relative to the current choice point to avoid regular actualizations when backtracking occurs. The value of the load register is the sum of the previous value plus the number of untried alternatives in the created choice point.

A great number of Prolog programs contain predicates with relatively small tasks. To attain good performances in the system it is fundamental to avoid sharing such kind of work. In YapOr the update of the load register is delayed. It is only updated after a certain number of *Call* instrutions are executed and whenever a sharing operation takes place, in which case it is reset to zero. A positive value in the load register indicates the scheduler that the worker has sharable work. By delaying the update of the load register, we want to encourage the workers to build up a reserve of local work and avoid over eager sharing.

Sharing Work

It is through the sharing work process that the parallel execution of goals becomes possible. When an idle worker finds a busy worker, it makes a sharing request and if a positive answer is received the sharing work process takes place. This process can be divided in four main steps:

Initial step: initialize auxiliary variables and compute the parts of the stacks to be copied.
Sharing step: realized by the busy worker, turn private choice points into public ones.
Copy step: copy computed segments from the busy worker stacks to the idle worker ones.
Installation step: update conditional variables present in the previous stacks of the idle worker to the bindings present in the busy worker stacks.

To minimize the overheads in sharing, both workers co-operate in the execution of the four steps (see figure 4). The idea is as follows: after a common initial step, the worker `P` with excess of load starts the sharing step while the idle worker `Q` starts the copy one. Worker `Q` copies the stacks from `P` in the following order: trail, heap and local stack. The local stack can only be copied after `P` finishes its sharing step. After the sharing step, `P` can help `Q` in the copy step, if it was not yet concluded. It copies the stacks to `Q` but in a reverse order. This scheme has proved to be efficient because it avoids some useless variables checks and locks. Finally, worker `P` returns to its Prolog execution while worker `Q` executes the installation step and restarts a new task from the recently installed work. If meanwhile, worker `P` backtracks to a shared node, it has to wait until `Q` finishes its installation step in order to avoid possible undoing of variable bindings.

Implementation of Cut

Prolog programs use the cut builtin, `!`, to discard unnecessary alternatives. On the other hand, or-parallel execution means that workers may go ahead and execute work that will later be pruned by cut. One says in this case that the work was *speculative*.

YapOr currently implements a simple mechanism for cut. The worker executing cut must go up the tree until either the root of the cut or a point where this some other worker to the left. While doing so it may find workers in branches to the right. If so, it sends them a signal informing their branches have been pruned. When receiving such a signal, workers will backtrack back to the shared tree and become idle workers again.

```
            Worker P          Signals          Worker Q
-----------------------------------------------------------------
.                                          .
Compute segments to copy                   Waiting accept signal
.                           ----accept---> .
Sharing step                               Copy trail ?
.                                          Copy heap ?
.                                          Waiting sharing_ok signal
.                           --sharing_ok--> .
.                                          Copy local stack ?
Help Q in copy step?                       .
.                           -----ready---> .
Waiting installation signal                .
.                           <-installation- .
Back to Prolog execution                   Waiting ready signal
.                                          .
Backtrack to shared node ?                 Installation step
Waiting ready signal                       .
.                           <----ready----- .
.                                          Back to Prolog execution
.                                          .
```

Fig. 4. Synchronizations between P e Q during the sharing work process.

Note that we may not be able to complete a cut if there were workers to the left, as these workers are capable of themselves pruning the cut we are in the process executing. One says in these cases the cut was left *pending*. In YapOr, pending cuts are only executed when the workers to the left finish their private work and backtrack into the public part of the tree. It will then be their responsibility to finish these cuts.

The existence of cuts and speculative work in the tree also affects scheduling. Muse implements a sophisticated strategy to avoid entering workers into scheduling work if there is non-speculative work available [AK92]. Further work is still necessary to make YapOr deal efficiently with this kind of work.

4 Performance Evaluation

The evaluation of YapOr was performed on a shared memory parallel machine, a Sun SparcCenter 2000 with 8 processors, 256 MBytes of main memory and a two level cache system. The system was running SunOS 5.6. The machine was otherwise idle while benchmarking. A number of benchmark Prolog programs commonly used to assess other parallel Prolog systems were used to assess YapOr. We measured the timings and speedups for each benchmark, compared YapOr's performance with that of Muse, and analyzed the parallel activities to identify potential sources of overhead.

Benchmark programs

All benchmarks find all the solutions for the problem. Multiple solutions are computed through "automatic backtracking on failure" after a solution has been found. The benchmark programs are:

8-queens2: this is a generate and test solution for the problem of placing 8 queens on a chess-board such that no two queens attack each other.
8-queens1: a more efficient algorithm to solve the 8 queens problem. In each step, the position for a new queen is determined by the current state of the chess board.
9-queens: this is the queens2 version without cuts for 9 queens.
5cubes: this solves the N-cubes (also called Instant Insanity) problem, with $N = 5$. It consists of stacking N colored cubes in a column, so that no color appears twice within any given side of the column (from [Tic91]).

puzzle: the goal is to place numbers in an hexagon such that the all sums of numbers in a diagonal will add to the same value.

ham: find all hamiltonian cycles in a graph.

nsort: naive sort algorithm. It sorts a list of 8 elements starting from the worst case (reverse order).

sm: solutions to the well-known Salt and Mustard puzzle.

db4, db5: complex database search queries in the chat system.

Timings and Speedups

To put the performance results in perspective we first compare YapOr's performance, configured with one worker, with the performance of Yap Prolog on the same machine. We would expect YapOr to be slower than Yap. YapOr must update the local work-load register, check for possible work-sharing solicitations, check for possible backtracking messages due to a cut operation, and perform small tests to verify whether the bottom-most node is shared or private. It was found that Yap is on average 13% faster than YapOr.

Table 1 presents the performance of YapOr with multiple workers. The table presents the execution times in milliseconds, for the benchmark programs, with speedups relative to the 1 worker case given in parentheses. The execution times correspond to the best times obtained in a set of 10 runs.

	Number of Workers					
Programs	1	2	4	6	7	8
puzzle	10.042	4.835(2.08)	2.316(4.34)	1.550(6.48)	1.339(7.50)	1.172(8.57)
9-queens	4.085	2.047(2.00)	1.026(3.98)	0.690(5.92)	0.596(6.85)	0.519(7.87)
ham	1.802	0.908(1.98)	0.474(3.80)	0.324(5.56)	0.281(6.41)	0.245(7.36)
5cubes	1.029	0.516(1.99)	0.260(3.96)	0.181(5.69)	0.170(6.05)	0.145(7.10)
8-queens2	1.063	0.606(1.75)	0.288(3.69)	0.202(5.26)	0.159(6.69)	0.149(7.13)
8-queens1	0.450	0.225(2.00)	0.118(3.81)	0.080(5.63)	0.072(6.25)	0.067(6.72)
nsort	2.089	1.191(1.75)	0.609(3.43)	0.411(5.08)	0.354(5.90)	0.315(6.63)
sm*10	0.527	0.274(1.92)	0.158(3.34)	0.128(4.12)	0.118(4.47)	0.115(4.58)
db5*10	0.167	0.099(1.69)	0.065(2.57)	0.068(2.46)	0.060(2.78)	0.061(2.74)
db4*10	0.133	0.079(1.68)	0.056(2.38)	0.055(2.42)	0.052(2.56)	0.060(2.22)
Σ	21.387	10.780(1.98)	5.370(3.98)	3.689(5.80)	3.201(6.68)	2.848(7.51)
Average		(1.88)	(3.53)	(4.86)	(5.55)	(6.09)

Table 1. YapOr execution times and speedups.

The results show that YapOr is effective in exploiting or-parallelism, giving effective speedups over execution with just one worker. The quality of the speedups achieved depends significantly on the amount of parallelism in the program being executed. The programs in the first group, *puzzle*, *9-queens*, *ham*, *5cubes*, *8-queens2* and *8-queens1*, have rather large search spaces, and are therefore amenable to the execution of coarse-grained tasks. This group shows very good speedups up to 8 workers. The speedups are still reasonably good for the intermediate group, programs *nsort* and *sm*10*. For the last group, with programs *db5*10* and *db4*10*, the speedups are rather poor and level off very quickly. The main reason for this to occur is the very low task granularities present in these programs.

It is interesting that in program *puzzle*, YapOr obtains super-linear speedups. This is probably due to lower miss rates, as the total cache size increases with the number of processors.

YapOr and Muse

Since YapOr is based on the same environment model as that used by the Muse system it is natural that we compare current YapOr's performance with that of Muse on a similar set of benchmark programs. We used the default compilation flags for Muse under SICStus, and also the default parallel execution parameters. The Muse version is the one available with SICStus Prolog release 3p6. Note that Muse under SICStus is a more mature system, and implements functionality that is still lacking in YapOr.

In table 2 we show the performance of Muse for the referred set of benchmark programs. YapOr benefits from the faster base performance of Yap, and manages to obtain about 20% better execution times on a single processor.

	Number of Workers					
Programs	**1**	**2**	**4**	**6**	**7**	**8**
puzzle	12.120	6.660(1.82)	3.720(3.26)	2.670(4.54)	2.230(5.43)	2.140(5.66)
9-queens	3.890	2.030(1.92)	1.110(3.54)	0.690(5.64)	0.630(6.17)	0.560(6.95)
ham	2.550	1.480(1.72)	0.820(3.11)	0.520(4.90)	0.520(4.90)	0.460(5.54)
5cubes	1.130	0.560(2.02)	0.280(4.04)	0.180(6.28)	0.160(7.06)	0.150(7.53)
8-queens2	1.350	0.690(1.96)	0.390(3.46)	0.270(5.00)	0.240(5.63)	0.220(6.14)
8-queens1	0.550	0.290(1.90)	0.160(3.44)	0.120(4.58)	0.110(5.00)	0.100(5.50)
nsort	2.650	1.450(1.83)	0.810(3.27)	0.550(4.82)	0.510(5.20)	0.450(5.89)
sm*10	0.670	0.360(1.86)	0.220(3.05)	0.170(3.94)	0.160(4.19)	0.150(4.47)
db5*10	0.190	0.110(1.73)	0.080(2.38)	0.070(2.72)	0.070(2.72)	0.070(2.72)
db4*10	0.160	0.090(1.78)	0.060(2.67)	0.070(2.29)	0.060(2.67)	0.070(2.29)
Σ	25.260	13.720(1.84)	7.650(3.30)	5.310(4.76)	4.690(5.39)	4.370(5.78)
Average		(1.85)	(3.22)	(4.47)	(4.90)	(5.27)

Table 2. Muse execution times and speedups.

Surprisingly, the results also show that YapOr obtains better speedup ratios than Muse with the increase in the number of workers. This is the case even though YapOr has better base performance. We suppose this might a problem with the default parameters used in Muse.

Parallel execution overheads

In this section we examine the various activities that take place during YapOr's parallel execution. In particular we timed the various activities a worker may be involved during execution in order to help determine which of those activities are causing a decrease in performance. The main activities traced are:

Prolog: time spent in Prolog execution, in verifying work-sharing requests and in keeping the workload register updated.
Search: time spent searching for a busy worker.
Sharing: time spent in the four phases of the work-sharing process.
Get-Work: time spent in obtaining a new alternative from a shared node. It includes backtracking to that node and locking to ensure mutual exclusion.
Cut: includes the time spent in the execution of a cut in the shared region and the time to move to another node whenever a cut operation takes place.

Table 3 shows the percentage of the execution time spent in each activity for two benchmark programs, one in the high-parallelism class and the other in the medium-parallelism class. The percentages were taken for a number of workers ranging between 1 and 8 workers.

Activity	Number of Workers 1	2	4	6	7	8
puzzle						
Prolog	100.00	99.95	99.56	99.20	99.02	98.68
Search	0.00	0.02	0.16	0.32	0.41	0.60
Sharing	0.00	0.02	0.17	0.32	0.38	0.50
Get-Work	0.00	0.01	0.10	0.17	0.19	0.23
Cut	0.00	0.00	0.00	0.00	0.00	0.00
sm						
Prolog	100.00	97.68	86.71	74.56	69.08	63.29
Search	0.00	0.81	5.02	11.50	13.85	16.87
Sharing	0.00	0.86	5.64	10.17	13.14	15.76
Get-Work	0.00	0.61	2.51	3.52	3.61	3.88
Cut	0.00	0.04	0.13	0.25	0.32	0.20

Table 3. Workers activities during execution.

The results show that when the number of workers increases the percentage of the execution time spent on the *Search* and *Sharing* activities have the biggest increase. This happens because of the increased competition for finding work which makes workers get smaller tasks and consequently increase the frequency by which they have to search for new work. The increased competition also makes for unexploited alternatives to be situated more and more in the shared part of the tree which increases the *Get-Work* activity.

The percentage of the execution time spent executing the cut predicate is minimal, indicating minimal overheads when executed in parallel.

One possible explanation for the decrease on the amount of parallelism is shown in table 4. This table shows the average number of tasks executed by one worker and the average size of a task. The size of a task may be defined as the average number of goals (that is *Call* instructions) executed within that task.

	Number of Workers 1	2	4	6	7	8
puzzle						
Tasks	1	213	1780	2763	3095	3813
Calls per task	171024	803	96	62	55	45
sm						
Tasks	1	53	188	267	308	362
Calls per task	7965	150	42	30	26	22

Table 4. Average number of tasks and *call* instructions per task.

The table clearly shows that increasing the number of workers decreases the granularity of the available parallelism. The consequence of this is that workers run out of work more quickly and therefore the activities related to work search, work sharing and getting work will become more important in the execution, causing overheads to increase. It is therefore no surprise that the performance for benchmark programs such as *salt-mustard* degrades significantly for larger numbers of workers.

5 Conclusions

We have presented the YapOr parallel Prolog system. This system uses the environment copying representation to implement Or-Parallelism. Benchmarks results show that the system is quite effective at exploiting Or-parallelism. The system has good sequential performance. It was able to achieve excellent speedups for applications with coarse-grained parallelism, and it performs better than Muse for the applications with medium parallelism. The good performance of YapOr was also explained by the fact that for most benchmarks YapOr spends most of its time executing reductions, not managing parallelism.

We have recently started researching on low-level simulation of YapOr. Our goal is to better understand the architectural parameters of the system, towards removing unnecessary bottlenecks, thus obtaining even better performance for medium-parallelism programs while facilitating porting to the new high performance and Distributed Shared Memory systems that are becoming available.

We are also actively working on extending YapOr to better support all builtins, and particularly speculative execution with cuts. YapOr is also the basis for our work to support parallel execution of tabulated logic programs [RSC97].

Acknowledgments

The authors would like to acknowledge and thank the contribution that Luís Damas gave with the Yap Prolog system. The first author is thankful to PRAXIS ("Sub-Programa Cência e Tecnologia do 2o. Quadro Comunitário de Apoio") for their financial support. This work was partially funded by the PROLOPPE project (grant PRAXIS/3/3.1/TIT/24/94).

References

[AK90a] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, December 1990.

[AK90b] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[AK92] Khayri A. M. Ali and Roland Karlsson. Scheduling Speculative Work in Muse and Performance Results. *International Journal of Parallel Programming*, 21(6), December 1992.

[BRS91] A. Beaumont, S. Muthu Raman, and P. Szeredi. Scheduling or-parallelism in Aurora with the Bristol scheduler. In *PARLE'91*, Lecture Notes in Computer Science. Springer-Verlag, April 1991.

[CS89] A. Calderwood and P. Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *The 6th International Conference on Logic Programming*, pages 419–435. The MIT Press, June 1989.

[CSC97] Manuel E. Correia, Fernando Silva, and Vítor Santos Costa. The SBA: Exploiting Orthogonality in AND-OR Parallel Systems. In *Proceedings of the 1997 International Logic Programming Symposium (ILPS'97), Port Jefferson, Long Island, NY.* The MIT Press, October 1997.

[DSCRA89] L. Damas, V. Santos Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual.* Centro de Informática da Universidade do Porto, 1989.

[GACH96] G. Gupta, Khayri A. M. Ali, Mats Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. March 1996. Preliminary Version.

[GJ93] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions on Programming Languages*, 15(4):659–680, September 1993.

[Kar92] Roland Karlsson. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, Stockholm, March 1992.

[LBD+88] Ewing Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, David H. D. Warren, A. Calderwood, P. Szeredi, Seif Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora Or-parallel Prolog System. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, Tokyo, November 1988.

[RSC97] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. On Applying Or-Parallelism to Tabled Evaluations. In *In Proceedings of the International Workshop on Tabling in Logic Programming, Leuven*, July 1997.

[Sil93] Fernando M. A. Silva. *An Implementation of Or-Parallel Prolog on a Distributed Shared Memory Architecture*. PhD thesis, Dept. of Computer Science, Univ. of Manchester, September 1993.

[Sin91] R. Sindaha. The Dharma Scheduler – Definitive Scheduling in Aurora on Multiprocessor Architecture. Technical Report, Department of Computer Science, University of Bristol, November 1991.

[Tic91] E. Tick. *Parallel Logic Programming*. The MIT Press, 1991.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, September 1983.