Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs

Kish Shen[1] and Manuel V. Hermenegildo[2]

[1] Department of Computer Science, University of Manchester, United Kingdom
[2] Department of Computer Science, T.U. of Madrid (UPM), Spain

1 Introduction and General Approach

We summarise our study of an important (but rarely examined) aspect of parallel execution in logic programming (LP): memory management, and the closely related issue of scheduling. We examine these issues in the context of implicit and-parallelism in non-deterministic programs, because it presents some of the most general problems (see [8] for justifications). This abstract is a highly condensed version of [8], and the reader is referred to that paper for details.

We use the "sub-tree" (or "multi-sequential" approach), where the computation is divided into "chunks" (**tasks**) which are worked on by individual processing agents (**workers**) cooperatively in parallel. To retain much of the sequential efficiency of state of the art sequential LP systems while achieving performance improvements through parallelism, each task is executed by the worker in much the same way as in a sequential implementation, except that parallel work can be made available for other idling workers during such execution.

Like in sequential LP systems, efficient memory management of parallel LP systems can be achieved by doing stack-based memory management, though parallelism introduces extra complications. Each worker has its own stack, and the stack of a sequential implementation thus becomes a distributed stack, with its state physically distributed across the various workers' stacks, but logically forming a single stack structure. For non-deterministic and-parallelism, where forking and joining of tasks can occur, additional data structures are needed to link and manage the physical distribution of the stack. These structures — **markers** — form the basis of the marker scheme, first introduced in [2]. Essentially, when each worker is working on a task, it uses its stack in much the same way as a sequential implementation would, but when it finishes the task and picks up another task to work on, a marker is allocated onto the stack to separate the memory areas used by the new task from those used by the old one and also to logically link and help manage the various sections of stacks used by the tasks. However, as pointed out in [2], the ability of Prolog to perform search, i.e. backtrack and try different alternative solutions, presents special problems for a distributed stack scheme – the problems of "trapped goals" and "garbage slots" (also referred to as "holes" in or-parallel systems): in a sequential system, backtracking causes the stack to shrink in size, and subsequent execution of an alternative causes the stack to grow again. By the same token, in a distributed stack scheme when a task backtracks, the stack section representing the task first shrinks and then grows again. However, because a worker's stack contains all the stack sections of the tasks it executed, the stack section that is being backtracked may not be at the top of the stack, and such stack sections are "trapped" by the stack sections following it on the worker's stack.

One way to avoid such problems, proposed in [2], is to restrict the scheduling of tasks so that a worker selects only tasks which would be executed in a sequential system later than the task it just completed ("appropriate" tasks). This ensures that backtracking can only occur in the topmost stack section of a worker but at the cost of having to determine appropriateness of tasks during scheduling and reducing parallelism. The reduction in parallelism can be partially overcome by allocating multiple stacks to each worker, but at the cost of a higher amount of (virtual) memory allocation, since each stack should be big enough to avoid high reallocation costs if it becomes full.

Because of these problems, alternative solutions were developed which lift the restriction on scheduling and allow backtracking in non-topmost stack sections. The purpose of this paper is to briefly sketch this newer approach which allows for flexible scheduling, and to present our experimental results which support our belief that the new approach is better than the previous proposal. For concreteness (and also because our experiments were performed on it), we shall discuss the scheme as implemented in the dependent and-parallel system DASWAM [5], which has many similarities to the scheme originally developed for the independent and-parallel &-Prolog [3]. A similar scheme has since also been implemented in the &ACE system [1].

2 A New Approach Allowing Flexible Scheduling

The new approach basically separates the shrinking from the subsequent regrowth of a stack section. In a sequential system, and in the original restricted scheduling scheme, backtracking causes the stack to shrink by popping items off the top of stack, and then when execution resumes, items are reallocated at the top of the same stack. In the new approach, shown in Fig. 1 if backtracking occurs in a non-topmost stack section the stack

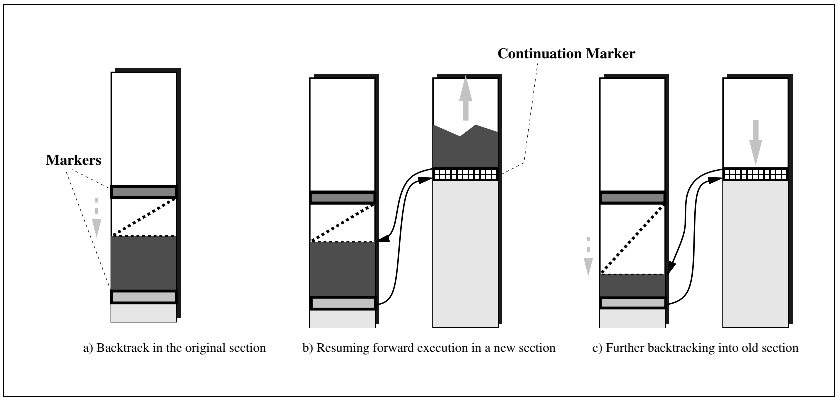

Figure 1. New Approach to Backtracking

is shrunk in that section, but subsequent regrowth occurs at a new location: the top of the stack of the worker that will execute the next alternative. This is done by introducing a specialised marker – the **continuation marker** – which links the partially shrunken old stack section and the new "regrowth" stack section together. Fig. 1a shows the shrinking of a non-topmost stack section during backtracking: the still allocated memory is shown in dark gray, and the deallocated memory is shown in white with a diagonal dotted line across it. After backtracking, a continuation marker is allocated in the stack in which the task is to be continued, and it carries sufficient information to allow backtracking to take place in the previous stack section, by having pointers into that previous stack section, as shown in Fig. 1b. If backtracking occurs in the new stack section, and it continues to the continuation marker, further backtracking is facilitated by following the pointer in the continuation marker to the old section. The old section is shrunk further, and then forward execution is resumed after updating the pointer in the continuation marker, as shown in Fig. 1c. If the whole section is shrunk to nothing, it can be unlinked entirely.

3 Evaluation of the impact of flexible scheduling

In order to evaluate the impact of the flexible scheduler for a fixed amount of virtual memory consumption we have compared the speedups obtained for the restricted scheduler and the flexible scheduler using the same number of stacks. We used ten programs, two of which can be considered as application type programs: 'Orsim', which is a simulator used for studying or-parallelism, and 'Ann', an automatic program paralleliser. Both of these programs are quite complex and contain over 1000 lines of source Prolog code. The other programs are simpler benchmark type programs, although some of them imply large executions. Excepting 'Qsort', they were chosen because they provide reasonably good speedups, as we expect the differences between the two schedulers to show up most clearly for programs with large speedups. 'Qsort' is used to see how a program with lower speedups would behave under the two schedulers.

We wish to compare the two schedulers in a machine independent way, so that our results are as generally applicable as possible, i.e. we are interested in the parallelism extracted by the two schedulers. To do this, we used the pseudo-parallel DASWAM system, which simulates parallelism, and obtains speedup figures which can be considered as the amount of parallelism available in the model, because it does not take hardware and software overheads for parallelism into account. Such idealised speedups are of great importance, because they allow for machine independent evaluation, better interpretation of actual parallel results, and even meaningful comparison between systems which exploit different types of parallelism (see [10, 11, 9, 7]). Note that such a comparison favours the restricted scheduler, as it does not take the cost of checking if a goal is suitable for execution or not into account in the computation of the speedups. In addition to comparing the two schedulers, we also ran the programs with an actual parallel implementation of DASWAM on a 26 processor Sequent Symmetry using the flexible scheduler. Nine of the ten programs show some differences in the speedups obtained by the two schedulers (bt_cluster was the only program that showed no difference), and these are shown in Fig. 2, along with the actual speedups from the parallel implementation. The actual speedups generally agree very well with the simulated speedups of the flexible scheduler, and in some cases they are so close that the lines representing the speedups are difficult to separate in the graphs. This shows that indeed the simulated speedups are meaningful, reaffirming the results from other studies [6]. For the nine programs, the results show that the flexible scheduler produces better, and in some cases much better, speedups than the restricted scheduler, for the same number of stacks. In practice, we expect the differences to be perhaps greater, because of any cost in checking for "appropriate goals". The figures also provide information on the speedups attainable with the restricted scheduler if an unbounded number of stacks were allowed: in this case the restricted scheduler would achieve at most the same speedup as the flexible scheduler

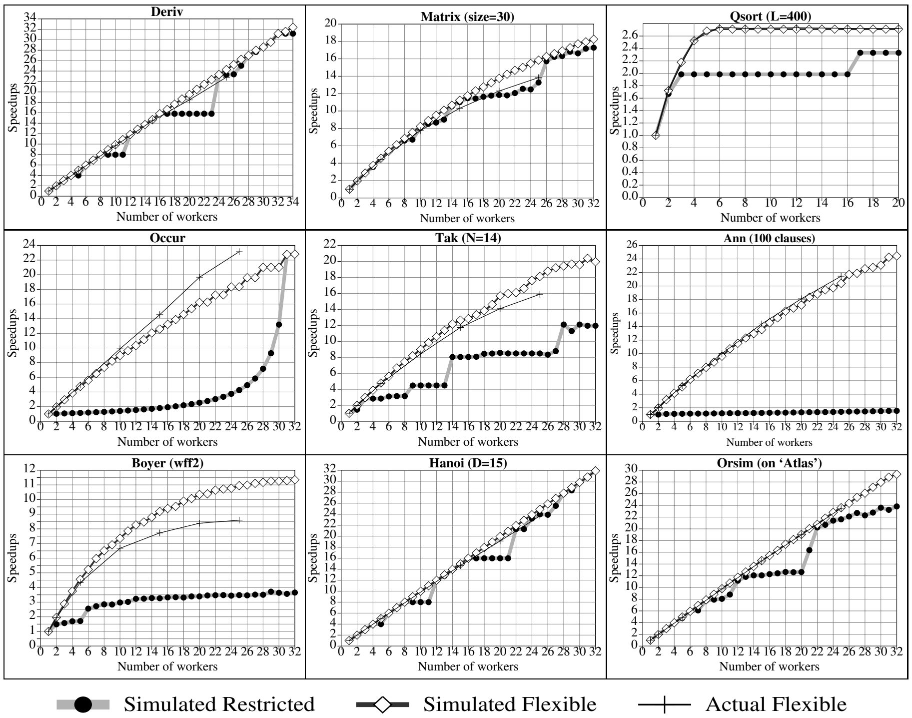

Figure 2. Comparison of speed-ups for the two scheduling strategies

(the difference limited by the cost of determining appropriateness of goals and changing stacks), and the number of stacks that would be required to achieve the this speedup can then be read from the graphs: it is equal to the number of workers needed for the restricted scheduler (with one stack per worker) to achieve this speedup. While in some cases (such as Matrix). this does not imply a high number of stacks, in others (such as Ann) this number can be very high (sometimes proportional to the depth of parallel recursions). Another disadvantage for the restricted scheduler is that we have found its speedup to be quite sensitive to the ordering of body goals in recursive clauses. There is insufficient space to present a more detailed analysis of the results; the interested reader is again referred to [8].

Prog	Seq.	par	flex10	res10
Boyer	7365×	24.0×	23.2×	23.4×
Orsim	609213×	1.42×	1.42×	1.42×
Tak	142×	346×	330×	291×
Hanoi	163372×	14.2×	13.8×	13.8×
Ann	46893×	1.15×	0.98×	1.05×
Occur	247680×	1.23×	1.64×	1.25×
Deriv	63058×	7.00×	6.81×	6.82×
Matrix	5870×	10.9×	10.9×	10.8×
Qsort	11223×	3.54×	3.47×	3.47×
Cluster	5024×	7.27×	7.07×	7.07×

Table 1. Memory Usage Comparisons

In addition to comparing the parallelism extracted, we also evaluated the memory performance of both schedulers in detail. Again, only a brief summary of this can be presented here. In Table 1 'Seq.' gives the total number of words of memory used by the (sequential) programs running on a sequential WAM. All the other columns are the amount of memory usages relative to this number: 'par' for the annotated program on one worker, 'flex10' for the annotated program on 10 workers with the flexible scheduler, and 'res10' for the annotated program on 10 workers with the restricted scheduler. Briefly, these results show that the flexible scheduler does not consume significantly more (physical) memory than the restricted scheduler (and it obviously consumes much less virtual memory, if the number of stacks is left unbounded). However, both schedulers sometimes consume significantly more memory than a sequential implementation. This memory consumption result, although shown to be smaller than that of other and-parallel systems using alternative memory management schemes [8], strongly justifies recently proposed optimizations for special cases such as deterministic computations [2, 4]. However, note that the memory consumption only tends to be much larger for small, benchmark type programs; the memory consumptions for the more realistic application-type programs (Orsim and Ann) are not excessive compared to the sequential consumption (Ann in fact consumes somewhat less memory in parallel because of parallelisation derived optimizations). The results also show that the memory consumption does not seem to increase significantly with more workers. Also, not shown by the summary here (but discussed in [8]), the memory usage is more evenly distributed across the workers for the flexible scheduler.

Finally, in [8] we also point out that the new data marker introduced can be applied to allow the efficient handling of the very general form of suspension that can occur in systems which support concurrency or combine several types of and-parallelism. We believe that the results are applicable to many and-parallel (and, also, or-parallel) systems.

References

1. G. Gupta, M. V. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP*, 1994.
2. M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, U. of Texas at Austin, 1986.
3. M. V. Hermenegildo and K. J. Green. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 253–268. The MIT Press, 1990.
4. E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Improving the Efficiency of Nondeterministic And–parallel Systems. *Computer Languages Journal*, 1996. In Press.[1]
5. K. Shen. *Studies of And/Or Parallelism in Prolog*. PhD thesis, U. of Cambridge, 1992.
6. K. Shen. Initial Results from the Parallel Implementation of DASWAM. Accepted at Joint International/Symposium of Logic Programming, 1996.
7. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *J. of Logic Prog.*, 1996. In Press.
8. K. Shen and M. V. Hermenegildo. Flexible Scheduling and Memory Management Scheme for Non-deterministic, And-parallel Execution of Logic Programs. Technical Report, 1995.
9. K. Shen and M. V. Hermenegildo. High-level Characteristics of Or- and Independent And-parallelism in Prolog. *Int. J. of Parallel Prog.*, 1996. In Press.
10. K. Shen and D. H. D. Warren. A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proc. Fourth SLP*, 1987.
11. P. Szeredi. Performance Analysis of the Aurora Or-Parallel System. In *Proc. NACLP*, 1989.

[1] Articles in press and technical reports are available at "`http://www.clip.dia.fi.upm.es/`".