&ACE: A High-Performance Parallel Prolog System

Enrico Pontelli, Gopal Gupta

Laboratory for Logic and DBs,
New Mexico State University
Las Cruces, NM 88003

Manuel Hermenegildo

Facultad de Informática,
Universidad Politécnica de Madrid,
Madrid, Spain

Abstract

In recent years a lot of research has been invested in parallel processing of numerical applications. However, parallel processing of Symbolic and AI applications has received less attention. This paper presents a system for parallel symbolic computing, named ACE, based on the *logic programming* paradigm. ACE is a computational model for the full Prolog language, capable of exploiting *Or-parallelism* and *Independent And-parallelism*. In this paper we focus on the implementation of the and-parallel part of the ACE system (called &ACE) on a shared memory multiprocessor, describing its organization, some optimizations, and presenting some performance figures, proving the ability of &ACE to efficiently exploit parallelism.

1 Introduction

Parallel processing of numerical problems has been an active area of research for almost two decades now [12, 17], resulting in the availability of parallelizing compilers and other tools that programmers can use without being experts in parallel processing. However, despite a certain amount of work in the area [11, 10], the same is not true of parallel processing of Symbolic and Artificial Intelligence (AI) applications: there are still few systems that are general enough and that programmers can use without being experts on parallel processing. In this paper we present a system that is a step towards the goal of building a general automatic tool for parallelizing symbolic and AI applications. Our approach is based on the *logic programming* paradigm and *Prolog* technology. A programmer wishing to parallelize his/her symbolic or AI program will code it in Prolog on a uniprocessor. Our compiler and runtime system will then run this Prolog program automatically in parallel. Our system supports the full Prolog language, including all its extralogical (cut), metalogical (var, assert, retract, etc.), and side-effect (read, write, etc.) predicates that are sensitive to the order of execution of the Prolog program.

Logic programming is a programming paradigm where programs are expressed as logical implications. Logic programming languages are suited for a wide range of applications, from compilers to databases to symbolic applications, as well as for general purpose programming. The most popular logic programming language is Prolog. An important property of logic programming languages is that they are single assignment languages. Unlike conventional programming languages they disallow destructive assignment and explicit control information. Not only does this allow cleaner (declarative) semantics for programs, and hence a better understanding of them by their users, it also makes it easier for an evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can be executed in any order without affecting the (declarative) meaning of the program. In particular, these operations can be performed by the evaluator *in parallel*.

An important characteristic of logic programming languages, thus, is that parallelism is easier to exploit in an *implicit* way. This can be done directly by the program evaluator (the runtime system) as suggested above, or, alternatively, it can also be done by a parallelizing compiler. The task of the parallelizing compiler is essentially to unburden the evaluator from making run-time decisions regarding which parts of the program to run in parallel. Note that the program can also be parallelized by the user (through suitable annotations). In all cases, the advantage offered by logic programming is that the process is easier because of the more declarative and high level nature of the language. Furthermore, the exploitation of parallelism at run-time or parallelization at compile-time can be done quite successfully in an automatic way and without requiring any input from the

user. Clearly, implicit exploitation of parallelism can in many cases have significant advantages over explicit parallelization.[1] In that sense, one can look in some ways towards Prolog for solving a new form of "(parallel) software crisis" that is posed to arise with the new wider availability of multiprocessors[2]—given systems, such as the one described in this paper, one can run Prolog programs written for sequential machines in parallel with no or minimal effort. For the rest of the paper we assume that the reader is familiar with Prolog and its execution model.

It must be pointed out that while the target application areas are Symbolic and AI applications, in fact, our system can also be used for parallel execution of general purpose programs. This stems from the fact that Prolog is an excellent language for writing problem solving (non-numerical) programs. This is borne out from some of the benchmarks we have used in the Performance Section (Section 4) of this paper.

Three principal kinds of (implicitly exploitable) control parallelism can be identified in logic programs:

1. *Or-parallelism* arises when more than one clause defines some predicate and a literal unifies with more than one clause head—the corresponding bodies can then be executed in parallel with each other, giving rise to or-parallelism.

2. *Independent and-parallelism* arises when more than one goal is present in the query or in the body of a clause, and it can be determined that these goals do not affect each other in the sequential execution–they can then be safely executed (independently) in parallel giving rise to (independent) and-parallelism.

3. *Dependent and-parallelism* arises when two or more non-independent goals are executed in parallel.

In this paper, we are mainly concerned with independent and-parallelism. Our purpose is to describe in full detail the independent and-parallel component of the ACE system, as it has been developed at the Laboratory for Logic, Databases, and Advanced Programming of the New Mexico State University collaboratively with Technical University of Madrid, Spain. In the rest of this paper we describe the operational semantics of the model of parallelism adopted, some effective optimizations, and finally some performance figures obtained on a Sequent Symmetry multiprocessor.

[1] This does not mean, of course, that a knowledgeable user should be prevented from parallelizing programs manually or even programming sequentially in a particular way that makes it possible for the system to uncover more parallelism.

[2] For example, affordable (shared memory) multiprocessor workstations are already being marketed by vendors such as Sun (Sun Sparc 10-2000), SGI (Challenge), etc.

2 ACE: An And-Or Parallel Execution Model

The ACE (And-or/parallel Copying-based Execution) model [7] uses stack-copying [1] and recomputation [6] to efficiently support combined Or- and Independent And-parallel execution of logic programs. ACE represents an efficient combination of Or- and independent And-parallelism in the sense that penalties for supporting either form of parallelism are paid only when that form of parallelism is actually exploited. Thus, in the presence of only or-parallelism, execution in ACE is *exactly* as in the MUSE [1] system—a stack-copying based purely Or-parallel system. In the presence of only independent And-parallelism, execution is *exactly* like the &-Prolog [8] system—a recomputation based purely And-parallel system. This efficiency in execution is accomplished by introducing the concept of *teams of processors* and extending the stack-copying techniques of MUSE to deal with this new organization of processors.

2.1 Or-Parallelism in ACE

ACE exploits Or-parallelism by using a stack-copying approach (like MUSE [1]). In this approach, a set of processing *agents* (processors in the case of MUSE, teams of processors in the case of ACE—as explained later) working in or-parallel maintain a *separate* but *identical* address space (i.e. they allocate their data structures starting at the same logical addresses). Whenever an *or-agent* (agents working in or-parallel are termed or-agents) $\mathcal{A}$ is idle, it will start looking for unexplored alternatives generated by some other or-agent $\mathcal{B}$. Once a choice point p with unexplored alternatives is detected in the computation tree $\mathcal{T}_\mathcal{B}$ generated by $\mathcal{B}$, then $\mathcal{A}$ creates a local copy of $\mathcal{T}_\mathcal{B}$ and restarts computation by backtracking over p and executing one of the unexplored alternatives. The fact that all the or-agents maintain an identical logical address space reduces the creation of a local copy of $\mathcal{T}_\mathcal{B}$ to a simple block memory copying operation (Figure 1). This whole operation of obtaining work from another agent is named *sharing* of or-parallel work.

In order to reduce the number of sharing operations performed (since each sharing operation may involve a considerable amount of overhead), unexplored alternatives are always searched starting from the bottom-

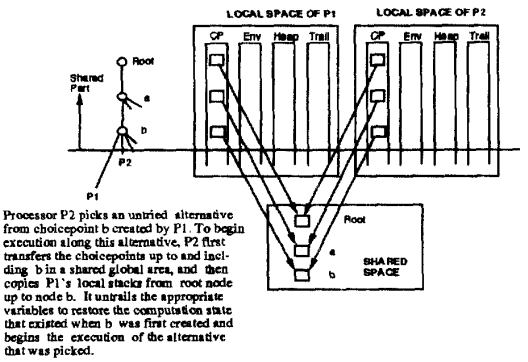

Figure 1: Stack-copying based Or-parallelism

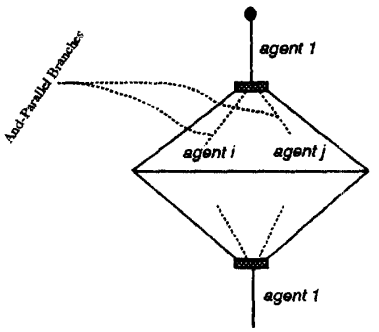

Figure 2: Computation Tree with And-parallelism

most part of the tree; during the sharing operation all the choice points in between are shared between the two agents (i.e. at each sharing operation we try to maximize the amount of work shared between the two agents). Furthermore, in order to reduce the amount of information transferred during the sharing operation, copying is done *incrementally*, i.e., only the difference between $\mathcal{T}_A$ and $\mathcal{T}_B$ is actually copied.

2.2 And-Parallelism in ACE

ACE exploits Independent And-parallelism using a recomputation based scheme [6]—no sharing of solutions is performed (at the and-parallel level). This means that for a query like `?- a,b`, where `a` and `b` are nondeterministic, `b` is completely recomputed for every solution of `a` (as in Prolog). Figure 2 sketches the structure of the computation tree created in the presence of and-parallel computation: a *parbegin-parend* structure is introduced, and the different branches are assigned to different agents. Since we are exploiting only *independent and-parallelism*, only independent subgoals are allowed to be executed concurrently by different *and-agents* (and-agents are processing agents working in and-parallel with each other). Dependencies are detected at run-time by executing some simple tests introduced by the *parallelizing compiler*. In ACE we have adopted the technique originally designed by DeGroot [4] and refined by Hermenegildo [9] (adopted also by &-Prolog [8]) of annotating the program at compile time with *Conditional Graph Expressions (CGEs)*. This will be explained in detail in a later section.

Since and-agents are computing just different parts of the same computation (i.e. they are cooperating in building one solution of the initial query) they need to have *different* but *mutually accessible* logical address spaces.

2.3 And-Or Parallelism in ACE

In ACE a clear separation is made between exploitation of or-parallelism and exploitation of and-parallelism. Processors in the multiprocessor system are divided into *teams of processors*. At a higher level, these teams of processors *execute in or-parallel* with each other (i.e., a team will take up only or-parallel work). At a lower level, i.e., within each team, processors in the team *execute in and-parallel* with each other (i.e., along an or-branch taken by a team, processors in that team will execute goals arising in that branch in and-parallel). Thus, the notion of *or-agent* is mapped to the notion of *team of processors* while the notion of *and-agent* is mapped to the notion of processors inside a team (i.e. each processor is an and-agent). This is illustrated in figure 3(i).
This organization into teams allows us to: (i) minimize the changes to be done to the basic and-parallel engine; (ii) clearly draw the boundaries between the different components of the system. Each processor in the ACE system is basically an and-parallel engine, capable of carrying on its own computation and interacting with a certain number of other processors (those belonging to the same team). The only new features that need to be added are the following: (a) a mechanism to keep track of the amount of or-parallel work produced by the computation of a team; (b) a mechanism to allow interaction of one team with another in order to guarantee synchronization and execution of *sharing operations*; (c) extended backtracking, allowing calls to the or-scheduler whenever a team backtracks over a shared choice point.

Or-parallel work is obtained by one team from another through the operation of *sharing*. A sharing operation involves copying (part of) the computation tree generated by a team to another team. The op-

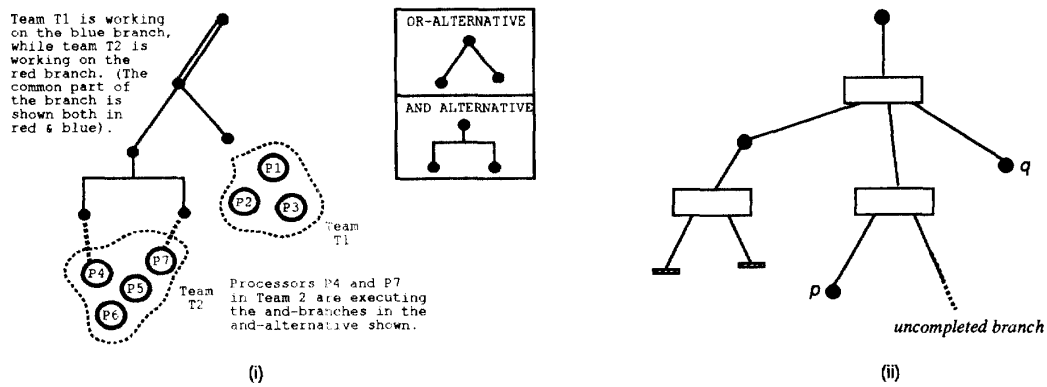

Figure 3: Processor Organization and Sharing Assumptions in the ACE system

eration is not straightforward since the computation tree is spread over the address spaces of the different processors belonging to the team. Furthermore, we want to perform copying incrementally, i.e., transfer only what is strictly necessary, and this selective operation is complicated by the arbitrary ordering of the different sections of computation on the stacks of the various team members. To make the whole process more effective, we have introduced in ACE some *sharing conditions* that a choice point should satisfy in order to allow its sharing with other teams (i.e., giving away its untried alternatives for or-parallel processing to other teams) . A choice point p satisfies the *sharing conditions* iff the whole computation on its left (in the and-tree exploited by the team) has already been completed. Figure 3(ii) shows an example of this: choice point p satisfies the conditions, while choice point q does not, since there is a branch on its left which has not completed yet (in the figure, a rectangular box represents an and-parallel call, each branch emanating from such a box corresponds to an and-parallel goal in the and-parallel call, and darkened circles represent choice points).

In ACE, during a sharing operation, only the choice points satisfying the sharing condition are actually taken into consideration. This simplifies both the copying operation (it is easier to detect which parts of the computation need to be transferred), the scheduling activity (we are guaranteed that everything on the left is terminated and successful and we do not have arbitrary intermixing of shared and private parts in the computation tree), and the management of side-effects and extra-logical predicates.

Different approaches to incremental copying have been studied and heuristics to choose the most appropriate in each situation have been developed. The interested reader is referred to [7] for a detailed discussion of this topic.

3 Independent and-parallelism

The objective of this paper is to illustrate the structure and the features of the and-parallel engine developed for the ACE system. In this section we explain in more detail the computational behavior of the and-parallel engine of ACE (named &ACE for the sake of simplicity). ACE has been implemented on top of *SICStus Prolog*, one of the most popular implementations of Prolog, and as such it inherits the basic structure of the SICStus WAM architecture [15] together with most of its features and optimizations. It should be also pointed out at the outset that our design of the and-parallel component of ACE is heavily influenced by RAP-WAM [9, 8].

3.1 Conditional Graph Expressions

A conditional graph expression (CGE for simplicity) is an expression of the form:

$$(\langle\, conditions \,\rangle \;\Rightarrow\; \mathtt{B}_1 \,\&\, \cdots \,\&\, \mathtt{B}_n)$$

where $\langle$ *conditions* $\rangle$ is a conjunction of simple tests on variables appearing in the clause which check for the independence of the goals and & denotes *parallel conjunction*. The intuitive meaning of a CGE is quite straightforward: if, at runtime, the tests present in *conditions* succeed, then the subgoals $\mathtt{B}_1 \,\&\, \cdots \,\&\, \mathtt{B}_n$ can be executed in and-parallel, otherwise they should be executed sequentially.

A standard Prolog program needs to be annotated with CGEs in order to take advantage of the and-parallel engines available. This process can be done manually by the programmer but is generally done automatically by specialized compile-time analysis tools (like the &-Prolog parallelizing compiler [2], which is

also an integral part of ACE).

3.2 Forward Execution

Forward execution of a program annotated with CGEs is quite straightforward. As long as CGEs are not encountered, forward execution is exactly the same as in SICStus WAM (i.e., standard Prolog execution). Whenever a CGE is encountered, the conditions are evaluated and, if the evaluation is successful, the various subgoals in the CGE are made available for and-parallel execution. Idle and-agents are allowed to pick up available subgoals and execute them. Only when the execution of those subgoals is terminated the continuation of the CGE (i.e. whatever comes after the CGE) is taken into consideration.

More precisely, when a CGE is met, a new descriptor for the parallel call (named *parcall frame*) is allocated, initialized, and all the subgoals but the leftmost one are loaded in a local work queue (*goal stack*) (the leftmost subgoal is directly executed by the same and-agent that created the parcall frame). The same processor performing the creation of the parallel call will eventually fetch and execute other unexecuted parallel subgoals from this parallel call, if necessary. An idle processor may pick work from the work queue of other processors. This will entail the identification of the subgoal to execute, the allocation of an initial data structure (to indicate the beginning of a subgoal execution—named *input marker*), the actual computation of the subgoal, and finally the allocation of a further marker (named *end marker*) to identify the completion point of the subgoal.

3.3 Backward Execution

Backward execution denotes the series of steps that are performed following a *failure*—due to unification or lack of matching clauses. Since an And-parallel system explores only one or-branch at a time, backward execution involves backtracking and searching for new alternatives in previous choice points. In ACE, where both Or- and And-parallelism are exploited, backtracking should also avoid taking alternatives already taken by other or-agents.

In the presence of CGEs, standard backtracking should be upgraded in order to deal with computations which are spread across processors. &ACE has a complete implementation of such a backtracking scheme.

As long as backtracking occurs over the sequential part of the computation no particular problems occur—we just use plain Prolog-like backtracking. Problem occurs when backtracking involves a CGE. Two cases are possible:

Backtracking occurs **inside** *one of the goals* $\mathbf{g_i}$ *of the CGE, and no solutions are found inside* $\mathbf{g_i}$: In this case we can observe that the whole CGE does not have any solution (since the subgoals are known to be independent). This allows the removal of the whole CGE and propagation of backtracking to the computation preceding the CGE itself.

Backtracking occurs in the continuation of the CGE (i.e., **outside** *a CGE), and there are no alternatives between the point of failure and the parallel call*: In this case backtracking should try to mimic Prolog backtracking, by searching for a new alternative moving from the rightmost goal of the CGE to the leftmost one. If a new successful alternative is found in the goal $\mathbf{g_i}$, then all the subgoals $\mathbf{g_j}$ (with $i < j \leq n$ where n is the number of parallel goals in the CGE) are re-executed in parallel. This is called *recomputation-based* and-parallelism, as mentioned earlier, since the subgoals on the right are completely recomputed for each new solution found on the left. If no successful alternatives are found in any of the subgoals of the CGE, then the whole CGE is removed and backtracking is propagated to the preceding computation.

This scheme allows us to obtain all the solutions of a CGE in the same order in which they are produced in a corresponding sequential execution (of course we are not considering the case in which some of the or-alternatives inside the CGE have been taken by some other or-agent for execution).

At the implementation level, backtracking is performed in the usual way, by moving downwards in the choice point stack and analyzing the data structures encountered. In case of &ACE this moving downwards is performed following the *logical* path—i.e. the logical "flow" of the computation—instead of the *physical* one, since in and-parallel system the two concepts are distinct (logical path and physical path coincide in a sequential execution). The main differences in this process occur whenever an end marker or an input marker is encountered. In the first case an outside backtracking phase has to be started (since backtracking is entering inside a completed parallel call), while in the second case the action depends on the current status of the computation: if at least one solution for each subgoal has already been found, then outside backtracking is in progress and backtracking needs to be propagated to the subgoal immediately on the right, otherwise inside backtracking is in place and the whole parallel computation can be discarded. Details of the implementation are omitted due to lack

of space, and they can be found in [5].

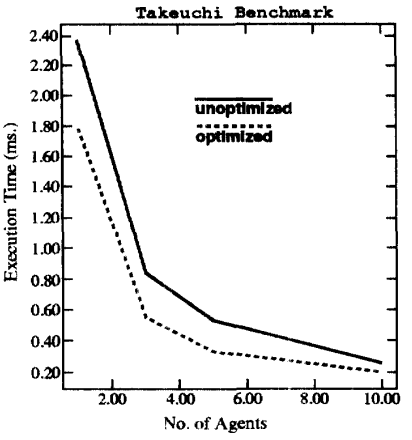

Figure 4: Unoptimized vs. Optimized Execution Time

3.4 Optimizations

Innumerable optimizations can be done in a recomputation-based and-parallel system like &ACE. An optimization that has been implemented in the current version of the system deals with taking advantage of deterministic computations. Many of the classical benchmarks proposed for and-parallelism involve the development of deep nestings of parallel calls, while the sequential subgoals (those which do not contain a further parallel call) are deterministic computations. The main idea is that once one of those deterministic computations has been completed, there is no need of keeping any data structure alive (since on backtracking there will not be any alternatives available). For this reason the allocation of the input marker is *delayed* until the first choice point/parcall frame is allocated (in a fashion similar to the shallow backtracking technique present in sequential systems [3]); if we reach the end of the computation without allocating any input marker, then the end marker itself is not allocated and we simply record the boundaries of the current trail section in the slot of the parcall frame relative to the current subgoal. On backtracking no action is needed for this kind of subgoals (the only required action is to unwind the trail section associated to their execution). This simple optimization allows saving time and space since the allocation of various data structures is avoided and the number of synchronization messages (during backtracking) is reduced.

Figure 4 shows the results of adopting the optimization on the Takeuchi benchmark. Even on a relative small computation time the difference between the unoptimized and the optimized version is significant (on a single processor we have observed an improvement of around 30% for certain benchmarks).

For further details on this and other optimizations, the reader is referred to [16].

4 Performance Results

The purpose of this section is to present the results obtained by executing some well-known benchmarks on &ACE. They range from simple test programs to actual applications. The results for the following benchmarks are initially reported: Matrix Multiplication, Quicksort, Takeuchi, Tower of Hanoi, Boyer (a reduced version of the Boyer-Moore theorem prover), Compiler (the Aquarius Prolog compiler from UC Berkeley that is approximately 2,200 lines of Prolog code), Poccur (a list processing program), BT_cluster (a clustering program from British Telecom, UK), Annotator (the annotator part of the ACE/&-Prolog parallelizing compiler that is about 1,000 lines) and, Simulator (a simulator for simulating parallel Prolog execution that is about 1,100 lines). Note that our suite of benchmarks does not just contain problems from the area of symbolic computing or AI. Rather, it also contains programs that are very "general" in nature, such as a compiler. This attests to the versatility of Prolog as a language and of the &ACE system that it is able to run in parallel a wide variety of programs.

Table 1 indicates, for each benchmark, the execution time (in ms.) and the speed-up obtained. The execution times are given in the format α/β, where α is the time obtained without the shallow-parallelism optimization and β is the time obtained using the optimized version. The speed-up figures are for the optimized execution. Some of the speedup curves are plotted in Figures 5(i) and 5(ii). The figures clearly indicate that the current implementation, even though not completely optimized, is quite effective. On many benchmarks, containing a sufficient amount of parallelism, the system manages to obtain linear speedups (like matrix multiplication and hanoi). With more processors in the multiprocessor systems we believe we should be able to obtain higher speedups, provided the program contains that much parallel work.

The largest benchmark is the Aquarius Prolog Compiler (approximately 2,200 lines of Prolog code). Note that for the *compiler*, *quicksort*, and *boyer* benchmarks, the speedup curve flattens out because at some point all available parallelism is exhausted (e.g., in the

Goals executed	&ACE agents 1	3	5	10
matrix_mult(30)	5598/5214	1954/1768 (2.95)	1145/1059 (4.92)	573/534 (9.76)
quick_sort(10)	1882/1536	778/632 (2.43)	548/455 (3.38)	442/373 (4.12)
takeuchi(14)	2366/1811	832/586 (3.09)	521/368 (4.92)	252/200 (9.06)
hanoi(11)	2183/1671	766/550 (3.04)	471/336 (4.97)	231/180 (9.28)
pderiv	—/5375	—/1840 (2.92)	—/1100 (4.89)	—/550 (9.77)
boyer(0)	9655/9290	5329/3829 (2.43)	3816/3199 (2.90)	2887/2687 (3.46)
compiler	—/29902	—/12522 (2.39)	—/6437 (4.65)	—/4801 (6.23)
poccur(5)	3651/3197	1255/1079 (2.96)	759/662 (4.83)	430/371 (8.62)
bt_cluster	1461/1343	528/480 (2.8)	345/312 (4.30)	/189 (7.11)
annotator(5)	1615/1422	556/475 (2.99)	392/322 (4.42)	213/187 (7.60)
simulator	—/3295	—/1232 (2.67)	—/827 (3.98)	—/536 (6.15)

Table 1: Unoptimized/Optimized Execution times in msec (Speedups are shown in parenthesis)

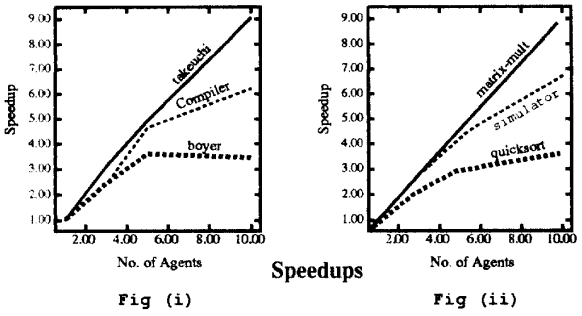

Figure 5: Speed and Speedups Curves for Selected Benchmarks on the Sequent Symmetry

case of compiler, the speed-up was measured while the compiler was compiling a program of 8 clauses, thus the maximum attainable speedup is 8; if a larger program were compiled higher speedups would be obtained). Our implementation incurs an average parallel overhead of about 10-15% over SICStus Prolog. Some of this parallel overhead is avoided by triggering optimizations mentioned earlier that are based on recognizing determinacy of goals. These optimizations yield, depending on the program, an improvement of 5% to 25% over the unoptimized version. Some improvement data is shown in Table 1 (each entry in Table 1 shows the time in milliseconds before the optimization and after the optimization; the number in parenthesis gives the speed-up obtained; for compiler and simulator benchmarks the unoptimized figure is not shown).

These results compare well with previous results reported for and-parallel systems [8] despite the fact that the ACE system includes the Or-parallel machinery and that the results reported in [8] assumed a static detection of determinacy to perform certain optimizations, while in &ACE these and other optimizations are performed dynamically (avoiding a potentially complex compile-time analysis).

5 Conclusions

This paper describes some of the most important features of the independent and-parallel component of the ACE system—a system which implicitly exploits both independent and- and or-parallelism from Prolog programs. We discussed the structure of the machine and the organization of the execution, placing emphasis on the new ideas and optimizations introduced in the design. We presented results for a comparatively large suite of benchmarks, some over two thousand Prolog lines long. Our results show that our system is well-suited for parallel execution of symbolic, AI, as well as general purpose applications coded in Prolog. These results also confirm our initial contention that ACE can exploit And-parallelism with the efficiency of competitive And-parallel only systems such as &-Prolog. Also, &ACE is arguably the most complete implementation of an and-parallel Prolog engine reported to date. It can support a much richer parallel backtracking behaviour than the original &-Prolog system [8] (to which the &ACE system owes a lot of its inspiration). It should be noted that designing and implementing such parallel backtracking behavior is a non-trivial task. Lin and Kumar [13] have also implemented an and-parallel system, called APEX, at the University of Texas at Austin. However, they have only tried comparatively simple benchmarks. Also, their system cannot handle full Prolog (which is important because not many practical applications can be efficiently written as pure logic programs). Another and-or parallel system, called Rolog, has been implemented by Ramkumar and Kalé [14] at University of Illinois [14]. Their system is based on a non-

backtracking model, thus it cannot take advantage of sequential Prolog technology. The sequential execution speed of our system is perhaps 5 to 10 times better than their's.[3] Rolog also does not support full Prolog.

References

[1] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.* MIT Press, 1990.

[2] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[3] M. Carlsson. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Sixth Int'l Conf. on Logic Prog.*, pages 3–16. MIT Press, June 1989.

[4] D. DeGroot. Restricted AND-Parallelism. In *Int'l Conf. on 5th Generation Computer Systems*, pages 471–478. Tokyo, Nov. 1984.

[5] M. Hermenegildo E. Pontelli, G. Gupta. &ace: The And-parallel Component of ACE (a progress report). Technical report, New Mexico State University, 1994.

[6] G. Gupta and M. Hermenegildo. Recomputation based Implementation of And-Or Parallel Prolog. In *Int'l Conf. on 5th Generation Computer Sys. '92*, pages 770–782, 1992.

[3] However, Rolog is able to considerably improve its performance by granularity control and parceling out large chunks of computation to efficient Sequential engines such as SICStus 0.6 and Quintus Prolog.

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

[8] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.*, pages 253–268. MIT Press, June 1990.

[9] 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, August 1986.

[10] L.N. Kanal V. Kumar, P.S. Gopalakrishnan, editor. *Parallel Algorithms for Machine Intelligence and Vision.* Springer Verlag, 1990.

[11] L. Kanal and C.B. Suttner, editors. *Proceedings of the Workshop on Parallel Processing for AI.* Technische Universität München, 1992.

[12] L. Lamport. The Parallel Execution of DO-loops. *Communications of the ACM*, 17:83–93, 1974.

[13] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth Int'l Conf. and Symposium on Logic Progr.*, pages 1123–1141, MIT Press, August 1988.

[14] B. Ramkumar and L.V. Kale. Machine Independent AND and OR Parallel Execution of Logic Programs. Part i and ii. *IEEE Transactions on Parallel and Distributed Systems*, 2(5).

[15] Swedish Institute of Computer Science. *Industrial SICStus Prolog Internals Manual*, January 1989.

[16] D. Tang E. Pontelli, G. Gupta. Determinacy driven optimizations of parallel prolog implementations. Technical report, New Mexico State University, 1994.

[17] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991.