Natural Language Multiprocessing: A Case Study

Enrico Pontelli, Gopal Gupta, Janyce Wiebe
Dept. Computer Science
New Mexico State University
{epontell,gupta,wiebe}@cs.nmsu.edu

David Farwell
Computing Research Laboratory
New Mexico State University
david@crl.nmsu.edu

Abstract

This paper presents two case studies of parallelization of large Natural Language Processing (NLP) applications using a parallel logic programming system (called "ACE") that automatically exploits implicit parallelism. The first system considered is *Artwork*, a system for semantic disambiguation, speech act resolution, and temporal reference resolution. The second system is *ULTRA*, a multilingual translation system. Both applications were originally developed in Prolog without any consideration for parallel processing. The results obtained confirm that NLP is a ripe area for exploitation of parallelism. Most previous work on parallelism in NLP focused primarily on parallelizing the parsing phase of language processing. The case studies presented here show that parallelism is also present in the semantic and discourse processing phases, which are often the most computationally intensive part of the application.

Content Areas: logic programming, NLP, AI system
Tracking Number: A208

Introduction

Logic programming has been for a long time one of the programming paradigms of choice for the development of NLP systems. Logic programming languages (e.g., Prolog) offer features such as backtracking, unification, and symbolic data representation which greatly facilitate the development of NLP applications (Pereira 1987). In recent years the use of declarative languages, such as Prolog and Lisp, for NLP applications has declined. This is mostly due to the increasing computational requirements of these applications, which were not satisfied by the older slow implementations of declarative languages. Even though implementations of these languages comparable in efficiency to imperative languages are available today (e.g., (Carlson & Gupta 1996)), past experience has discouraged designers of NLP systems from using them.

The computational demands of today's NLP systems have reached the level where they are challenging for even the most efficiently implementable programming paradigms, including imperative languages. Exploitation of parallelism is an important technique, especially given that multiprocessor machines (e.g., dual and quad Pentium-based PCs) are readily and cheaply available in the market today. Logic programming languages have a distinct advantage over imperative languages with regards to parallelism: their mathematical semantics allow for *automatic* or *semi-automatic* exploitation of parallelism, thus relieving the programmer from most of the intricacies of parallel programming and debugging. One of the goals of this paper is to make the NLP community aware of the recent advances made in implementation and parallelization technology for logic programming, and their potential benefits to NLP systems. The current implementations of logic programming systems are comparable in execution efficiency to any other programming paradigm. We hope that all of these attractive features—the ability to automatically or semi-automatically exploit parallelism, realization of non-determinism, support of symbolic data representation, and efficiency comparable to imperative paradigms—will convince NLP system designers to reconsider logic programming as an excellent implementation choice for many NLP applications. Given that parallelism can be exploited automatically, existing Prolog-based NLP applications can also be parallelized, by porting them with little or no effort to parallel systems. Fast parallel implementations of Prolog are currently available (e.g., (Ali & Karlsson 1990)) or about to be released into the public domain, including the ACE system (Gupta *et al.* 1994) used in this work.

In this work we present two case studies in which two large NLP applications, independently developed with no goal of parallelization in mind, have been studied and parallelized using the ACE parallel Prolog system. The parallelization effort was very limited in both cases and lead to excellent performance and speedups for both applications. In the rest of this paper we give a brief description of parallel logic programming and the ACE system followed by an overview of the two NLP applications considered (Artwork and ULTRA). Finally, we analyze the performance results obtained.

Parallelism in Logic Programming

Logic programming is a programming paradigm in which programs are expressed as logical implications. 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 a runtime 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 *implicitly in parallel.* Parallelization can be done directly by the runtime evaluator as suggested above, or, alternatively, it can also be done by a parallelizing compiler. The task of the parallelizing compiler is 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.

Three principal kinds of (implicitly exploitable) control parallelism can be identified in logic programs: *(i) 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. Or-parallelism is thus a way of efficiently searching for solutions to the query, by exploring alternative solutions in parallel. *(ii) Independent and-parallelism (IAP)* arises when more than one goal is present in the query or in the body of a clause, and at runtime these goals to do not compete for any unbound variable. *(iii) Dependent and-parallelism (DAP)* arises when two or more goals of a clause access common variables and are executed in parallel.

The ACE System

The ACE (And-Or/Parallel Copying-based Execution) model (Gupta *et al.* 1994; Pontelli 1997) uses stack-copying (Ali & Karlsson 1990) and recomputation (Gupta *et al.* 1994) 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. This efficiency in execution is accomplished by introducing the concept of *teams of processors* and extending the stack-copying techniques to deal with this new processor organization.

Or-Parallelism in ACE
ACE exploits or–parallelism by using a stack copying approach (like MUSE (Ali & Karlsson 1990)). 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.

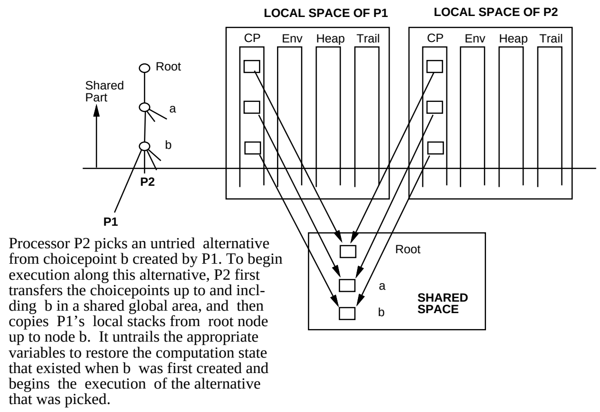

Figure 1: Stack-copying based or-parallelism

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-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 $\mathcal{T}_\mathcal{A}$ and $\mathcal{T}_\mathcal{B}$ is actually copied.

And-Parallelism in ACE ACE exploits IAP using a recomputation based scheme—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*. ACE adopts the technique originally designed by DeGroot (DeGroot 1984) and refined by Hermenegildo (Hermenegildo & Greene 1990) of annotating the program at compile time with *Conditional Graph Expressions (CGEs)*:

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

where $\langle\, conditions \,\rangle$ is a conjunction of simple tests on variables appearing in the clause that verifies whether the arguments share any variables with arguments of other goals, and & denotes *parallel conjunction*. Intuitively, if the tests present in *conditions* succeed, then the subgoals $\mathtt{B}_1 \,\&\, \cdots \,\&\, \mathtt{B}_n$ can be executed in and-parallel, else they should be executed sequentially.

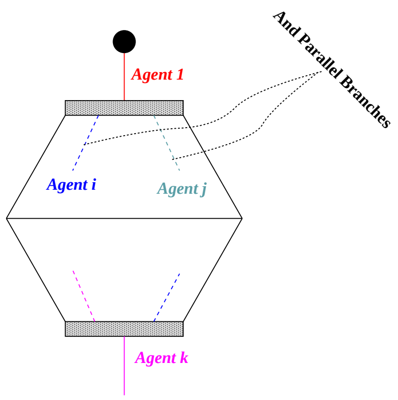

Figure 2: Computation Tree with and-parallelism

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* address spaces.

When a CGE is seen, 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, the actual computation of the subgoal, and finally the allocation of further structures to identify the completion of the subgoal. *Backward execution* denotes the series of steps that are performed following a *failure*—due to unification or lack of matching clauses. 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.

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 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)—as in figure 3. Different approaches to incremental copying and heuristics have been developed (Gupta *et al.* 1994).

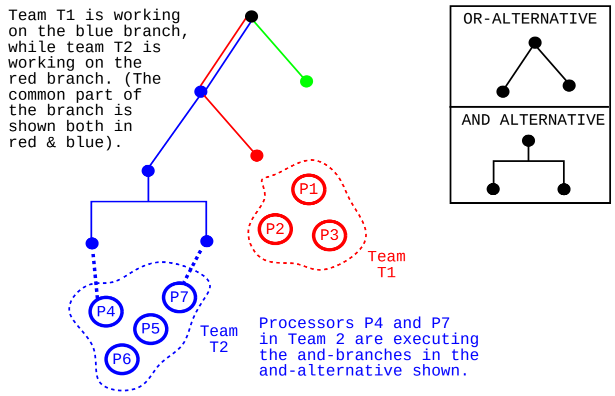

Figure 3: Processor Organization in the ACE system

ACE has shown remarkable performance results on a large selection of programs (Figure 4 presents the speedup curves obtained on some benchmarks). Furthermore, the parallelization overhead is extremely low, on average 5 to 10% w.r.t. sequential SICStus Prolog, on which the ACE engine is based.

NLP Applications

The field of NLP has changed dramatically over the last few years. The pressure from the users community has pushed NLP to increase the amount of effort towards the development of specific, practical applications and large-scale language processing systems. This has created an increasing need for efficient implementation schemes for many of the theoretical concepts developed in the past. NLP is probably one of the application fields with the highest computational requirements. Problems tend to generate high levels of nondeterminism and ambiguity, making exhaustive ap-

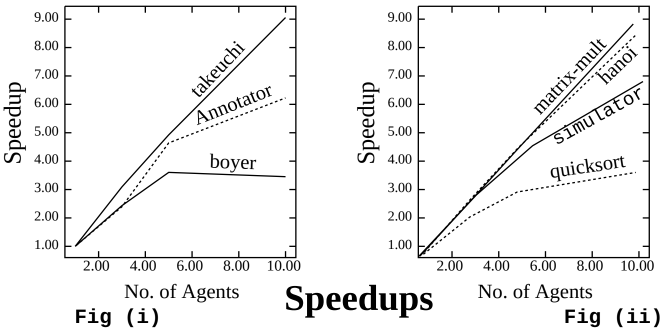

Figure 4: Speedups in ACE

proaches entirely not feasible. This forces researchers to focus on smaller problems or operate under more conservative assumptions. Our thesis in this paper is that by resorting to exploitation of parallelism many of these conservative assumptions can be relaxed, as well as larger problems solved. We take advantage of parallelism implicit in NLP applications to achieve efficient execution as well as improve the precision of the solutions obtained (by exploring a larger portion of the search space). Our experience with parallelizing the Artwork and the ULTRA systems and the results obtained confirm our thesis. The elegance and the semantic clarity of logic programming not only allowed simplification of the software development process, but also permitted automatic exploitation of parallelism; such exploitation of parallelism and execution speedups would not have been possible if the code were written in an imperative programming language, since for such languages automatic parallelization is extremely difficult.

Artwork

When people write or converse, they leave much of what they are communicating implicit, leaving the listener or hearer to fill in the missing information by considering the surrounding context. Consider, for example, understanding the pronouns in the sentence "She was there that day." Who was where on what day depends on what was said previously. The tasks of the discourse processing component of a NLP system involve recovering information that is implicitly communicated in the text or dialog. Such a component performs these tasks by considering information across sentence boundaries, rather than processing each sentence in isolation.

Only recently have major efforts been invested in empirical investigations of computational theories of discourse processing (see, for example, the special issue of *Computational Linguistics* devoted to empirical studies in discourse (23(1) 1997)). The Artwork project is one such NLP project, funded by the Department of Defense. It targets *scheduling dialogs*, dialogs in which the participants schedule a meeting with one another. A fully automatic rule-based system was developed that performs many kinds of natural language disambiguation, including *semantic* disambiguation, and two types of discourse disambiguation: *speech-act* and *temporal reference* resolution.

The input to the system is the output of a semantic parser developed as part of the Enthusiast speech-to-speech machine translation project at CMU (Levin 1995); it determines, among other things, what type of event each utterance is about, and who the participants are. The output of the parser is ambiguous, so must be disambiguated by the Artwork system.

The first of the discourse ambiguities addressed involves a prominent view of language as goal-oriented behavior. Under this view, which is adopted in this project, utterances are produced by actions that are executed with the goal of having some particular effect on the hearer. These actions are called *speech acts*. The task of an understanding system is to recognize which speech acts the speaker is performing with his or her utterances (Reithinger & Maier 1995; Rosé *et al.* 1995; Wiebe et al. 1996). Consider the utterance "2 to 4" (dos a cuatro), a common type of utterance in the scheduling dialogs. The speaker might be *suggesting* that they meet from 2 to 4; they might be *confirming* that 2 to 4 is the time currently being discussed; they might, with the right intonation, be *accepting* 2 to 4; and so on. The other kind of discourse ambiguity addressed is temporal (Wiebe et al. 1997). The Artwork system tracks the times being talked about, determining implicit contextual information. For example, when a speaker refers to "2 to 4 am", Artwork looks back at the previous utterances, and decides which day, date, and month are being referred to. This involves a search for the best possible candidate which fits the constraints generated by the previous utterances. Temporal reference resolution is the primary focus of the current Artwork system. Wiebe et al. (1997) present the results of the system performing this task on unseen, held-out test data, taking as input the ambiguous output of the semantic parser (which itself takes as input the output of a speech recognition system (Levin 1995)). The system performs well, and comparable results on similar tasks have not been published elsewhere.

The system parallelized in this work is an earlier version which performed speech-act and temporal reference resolution, but not semantic disambiguation. It includes the core architecture and speech-act and temporal-reference resolution rules of the current system. We expect the results obtained in this study to transfer easily to the latest versions of Artwork.

ULTRA

ULTRA (Universal Language TRAnslator) is a multilingual, interlingual machine translation system. It can currently translate between five languages (Chinese, English, German, Spanish, and Japanese) with vocabularies in each language based on about 10,000 word senses. The multilingual system is based on a language-independent interlingual representation (IR) (Farwell & Wilks 1991) for representing expressions as elements of linguistic acts of communication (e.g., asking questions, describing the world, promising that things will get done, etc.). Translation can be viewed as the use of the target language to express the same act as the one expressed in the source language. The IR is then used as the basis for analyzing or for generating expressions as elements of such acts in each of the languages (Farwell & Wilks 1991).

Each individual language system is independent and has its own rules to associate the appropriate IRs to each expression of the language. The different language components interact exclusively via IRs—thus allowing, for example, a clear system design and making the addition of new languages very easy without unpredictable effects on the rest of the system. It also gives freedom to the implementor to choose the class of grammars and the parser (s)he prefers. Currently most of the language components are implemented as context-free grammars with complex categories.

The system uses relaxation techniques (grammatical relaxation, semantic relaxation, and structure relaxation) to provide robustness by giving preferred or "near miss" translations (Farwell & Wilks 1991). In addition, the adoption of language-independent semantic and pragmatic procedures allows, given a context, the selection of the best IR from the set of possible IRs for a given expression (Slator 1988). The use of Prolog and Definite Clause Grammars (Pereira & Warren 1980) allowed the design of a highly declarative and perfectly bidirectional application.

The current prototype produces word, phrase, or sentence level translations and handles most basic declarative, interrogative, and imperative structures, including conjoined and subjoined constructions, while dealing with various types of sense disambiguation and structurally dependent anaphora and ellipsis.

Experimental Results

Parallelization of Artwork

The parallelization of Artwork was performed semi-automatically. The ACE compiler was used to identify potential sources of parallelism. Additionally, the rich output of the ACE static analyzer (Pontelli et al. 1997) (e.g., sharing information) allowed us to identify features of the program that were limiting the parallelism exploitable. Very few hand-modifications of the original code were needed to considerably improve the speedups achieved, as discussed in the next subsection.

Ambiguity in NLP gives rise to a "combinatorial explosion" of possible interpretations. Consequently, Artwork may take up to a couple of hours to process a dialog (Wiebe et al. 1996). Artwork offers a great deal of inherent parallelism to exploit, including or-parallelism and both DAP and IAP.

To process each utterance, the system applies all rules in its knowledge base. Each rule that matches the utterance fires, producing a partial representation. The rules are not all competitors; some of them target one ambiguity while other rules target others. All possible maximal merging of the results are formed, resulting in the set of interpretations the system chooses among to be the representation of the utterance. The application of the rules is realized through a subgoal of the form:

```
findall(R,int(PrevI,PrevUt,CurI,T_Ut,_,_,R),List)
```

where the `int` predicates is defined as:

```
int(PrevI,PrevUt,CurI,CurUt,CF,Rule,Res):-
    int01(PrevI,PrevUt,CurI,CurUt,CF,Rule,Res).
int(PrevI, PrevUt,CurI,CurUt,CF,Rule,Res):-
    int02(PrevI,PrevUt,CurI,CurUt,CF,Rule,Res).
....
```

The rules are independent of one another, which allows their application to the current utterance in parallel. The `findall` was unfolded giving rise to instances of IAP. The unfolding into and-parallelism was preferred to an or-parallel execution of `findall` to reduce the amount of parallel overhead.

The ACE compiler was capable of detecting a good source of DAP in the merging process, where partial results become known, by performing in parallel the separate merging of two incompatible partial representations with the others. The parallel annotation produced for this phase is:

```
buildComplete(ILTlist,PrevI,Ut,PrevUt,Comp,CList) :-
  getfirst(ILT,ILTlist,R),
  normalizefinal(ILT,CurI),
  (dep([PList]) →
     buildPartials(PrevI,CurI,PrevUt,Ut,PList) &
     merge(PList,CurI,NewComp)),
  append(Comp,NewComp,NList),
  buildComplete(R,PrevI,Ut,PrevUt,NList,CList).
```

The two subgoals separated by '&' can be executed concurrently, and the variable annotated with `dep` represents the communication channel between the two parallel threads. Unfortunately, the presence of the `append` subgoal represents a barrier which does not allow the results to be immediately propagated to the continuation of the computation. This problem was tackled by hand-modifying the code to allow exploitation of parallelism between the processing of the different elements of the input list `ILTlist`. Additional smaller sources of parallelism are present in various parts of the program, e.g., IAP between different actions in the body of each rule.

From the point of view of or-parallelism, we have identified various components of the systems where a local search is performed, making them suitable for or-parallel execution. For example, table 1 illustrates the improvement in execution of the selection of prediction phase. Globally, or-parallelism did not produce excessive speedups, due to the shallow searches produced by the Artwork benchmarks available. Furthermore, or-parallelism in Artwork was negatively affected by frequent use of various side-effect predicates and cuts.

Query	ACE Agents			
	1	2	3	4
Sentence_1	4810	3620	1623	1503

Table 1: Parallel prediction (Sun Sparc, times in ms.)

Speedups of the Artwork system from exploiting and-parallelism was considerably more satisfactory. The initial speedups observed were modest, as illustrated in the first two lines of table 2. These modest speedups, as discussed above, are not due to a lack of parallelism inherent in the application, but to the occasional use of some partially non-declarative constructions. Very few hand-coded modifications (driven by the results obtained from the static analyzer) were performed to make the code more declarative. These modifications should not be regarded as "tinkering" with the program to elicit better speedups; rather, they were simply attempts to replace extra-logical built-ins and features of Prolog with declarative ones. The more declarative and semantically elegant the program the more parallelism there is to exploit. The excellent results obtained from this modified version of the system can be seen in the last two lines of table 2. The speedups are very good, as shown in fig. 5. The parallel overhead recorded is less than 10%.

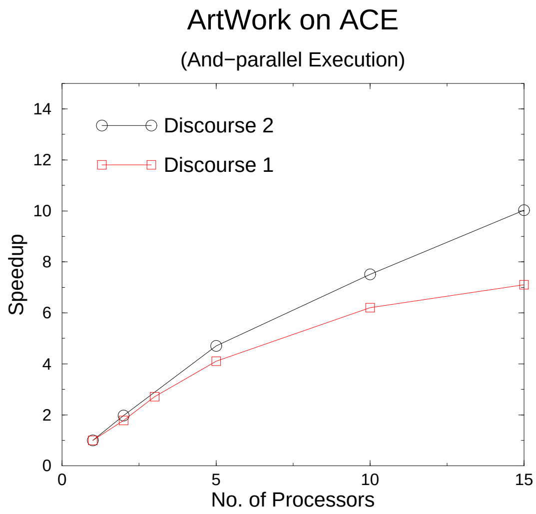

Figure 5: Speedups for and-Parallel Artwork

As part of future work, we plan to explore the use of or-parallelism to relax limitations in the number of input alternatives considered by the current system (recall that the input to the current system is ambiguous). The current limitations may lead to the exclusion of correct interpretations. We expect that processing alternative interpretations in parallel will allow the system to consider additional candidate interpretations, increasing the chance of a correct interpretations.

Parallelization of ULTRA

The parallelization of the ULTRA system was performed completely automatically (note that the Ultra system was developed long before the development of the ACE system was begun).

The analyzer was quite successful in detecting parallelism in the program. At the highest level, IAP was exploited by allowing concurrent translation of successive phrases belonging to the source text. The translation of each phrase was marked as a potential source of DAP by the dependent and-parallel analyzer (Pontelli et al. 1997), as illustrated below:

```
ctrans(Src_lg,Trg_lg,In,Out) :-
 ((dep([I_rep])-> (analyse(Srg_lg,In,I_rep) &
                   generate(Trg_lg,I_rep,Out));
     Out_String = "< unable to translate >").
```

where '&' denotes a parallel conjunction and the `dep` annotation identifies the source of dependency (*shared variable*). Similarly, the process of analyzing of each individual sentence to produce the corresponding IR contains considerable amounts of both DAP and IAP, as illustrated in the example below:

```
e_analyse(String,Struc) :-
  dep([List]) -> (prep_list(String,List) &
         e_prdctn(_A,_B,_C,Struc,List,[])).
```

All the lower level rules which identify the various sentence structures provide additional IAP:

```
e_prdctn(indpnt,conclusion,
     nil,[prdctn,[type,indpnt],
     [class,conclusion],[form,nil],
     [closing, Conclusion], Name],In,Out) :-
                split_input(In,In1,In2,Out),
                (esign_off(In1,Conclusion) &
                 ep_name(In2,human,Name).
```

IAP emerged consistently also in the the second phase of the translation, where IRs are mapped to sentences in the target language. The rule below is an example:

```
s_prdctn(indpnt,Pc,fin,
     [prdctn, [type, indpnt],
     [class, Pc], [form, fin],P1,P2]):-
 (s_prop1(dpnt,adv,fin,_M,_S_a1,_S_g1,P1) &
  s_prop1(indpnt,Pc,fin,_Pm,_S_a2,_S_g2,P2)),
 (Pc = dcl; Pc == imp; Pc = int).
```

No significant or-parallelism was detected in these examples (as indicated, for example, in table 3).

Query	ACE Agents		
	1	2	4
English to Chinese	322669	290402 (1.11)	251682 (1.29)

Table 3: Or-parallelism in ULTRA (Sequent, ms.)

Table 4 presents the performance figures achieved running the parallelized version of ULTRA on three separate examples, translating the same text from English to three different target languages, Chinese, Spanish, and German. In most cases the improvement in execution time due to exploitation of parallelism is

Goals executed	ACE agents						
	1	2	3	5	10	15	18
Discourse 1	78300	62060 (1.26)	51679 (1.52)	50389 (1.55)	50192 (1.56)	50190 (1.56)	50190 (1.56)
Discourse 2	48570	31234 (1.56)	21539 (2.25)	21500 (2.26)	20935 (2.32)	20650 (2.35)	20622 (2.36)
Modified 1	73594	41345 (1.78)	27156 (2.71)	17950 (4.1)	11870 (6.2)	10365 (7.1)	10301 (7.14)
Modified 2	46529	23639 (1.97)	15919 (2.92)	9889 (4.71)	6200 (7.5)	4640 (10.03)	4409 (10.55)

Table 2: Execution Times for Artwork (Sequent Symmetry, times in ms.)

excellent, confirmed by the speedup curves (fig. 6). Also in this case, the parallel overhead is extremely low (around 5%).

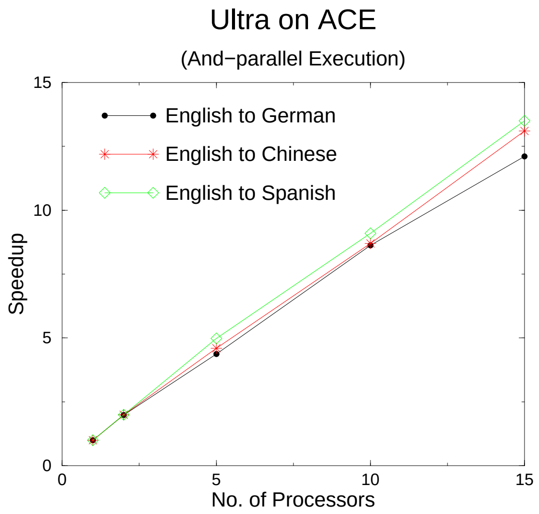

Figure 6: Speedups for ULTRA System

The automatic annotation performed by the ACE compiler was rather slow, due to the size and organization of the application—a single module of over 35, 000 lines of Prolog code. A modular reorganization of the code and the use of a newly developed incremental analysis technique (Hermenegildo *et al.* 1995) will improve the speed of annotation. Even more, the annotator is written in Prolog, so the annotation process itself can be parallelized.

Comparison with Other Work

Experience shows that NLP applications are highly parallel in nature. Although considerable research has been proposed in using parallelism for NLP (see (Adriaens & Hahn 1994) for a review of approaches to parallel NLP), most of the approaches are based on ad-hoc and hand-made parallelization of NLP applications.

There has been research in the past in parallelizing the parsing phase of NLP (Devos et al. 1988), including some that use a logic programming approach (Matsumoto 1986; Trehan & Wilk 1988). More recently, the Eu-PAGE systems (Manousopoulou et al. 1997) has been proposed, a parser generator capable of producing parallel parsers (based on PVM). However, not much parallelism has been extracted from other phases of NLP systems. The main exception to this is the PUNDIT system (Hirschman et al. 1988) which exploits (only) or-parallelism from an NLP system coded in Prolog running on a simulated parallel environment. The PUNDIT system is about 3,000 lines long. Thus, our proposed work can be seen as taking the work done on the PUNDIT system further and exploiting both or- and and-parallelism from NLP programs. It should be noted that almost all work on parallel NLP (including the PUNDIT system) relies on explicit parallelization of the NLP system by the programmer. Our research results in this respect are distinctive in that: (i) we exploit parallelism from NLP applications implicitly or with limited programmer intervention, and (ii) we have been successful in parallelizing a significantly larger NLP application (it should be noted that in the PUNDIT system, the developers of the original system were the ones who did the parallelization; the same is true of other parallel NLP systems), whereas in our case system development and parallelization was decoupled.

Conclusions and Future Work

In this paper we presented two case studies of parallel execution of large NLP applications—Artwork and ULTRA. In both cases we have taken advantage of the fact that the applications have been developed using logic programming, and logic programming is particularly suited to both automatic and semi-automatic exploitation of parallelism inherent in the application. In both cases we relied on the use of automatic compile-time analysis and run-time tools to extract parallelism (either directly or to supply information to the programmer to achieve this goal). Thus, the exploitation of parallelism required only a limited understanding of the behaviour and structure of the programs.

Our results show that the use of logic programming allows one to go beyond just the parallelization of the parsing phase of NLP systems, which has been the sole focus of most of the previous efforts. That is, it also allows the exploitation of parallelism from the semantic and discourse processing phases—which are typically the more computationally intensive.

In the case of ULTRA the goal of parallelization was quickly achieved with limited effort, thanks to the clean and declarative programming style adopted by the programmers who developed the system, necessitated by the requirement that computations be reversible (Farwell & Wilks 1991). In the case of Art-

Goals executed	ACE agents						
	1	2	3	5	10	15	18
English to German	562740	282339 (1.99)	190115 (2.96)	128509 (4.38)	65283 (8.62)	46507 (12.1)	37768 (14.9)
English to Chinese	322669	160100 (2.0)	108642 (2.97)	70145 (4.6)	37088 (8.7)	24631 (13.1)	20817 (15.5)
English to Spanish	91519	45756 (2.0)	30505 (3.0)	18370 (4.98)	10057 (9.1)	6779 (13.5)	5684 (16.1)

Table 4: Execution Times for ULTRA (Sequent Symmetry, times in ms.)

work the system had been developed following a more "imperative" approach in the code organization. We invested some effort in reorganizing some parts of the code and this proved effective in increasing the amount of and-parallelism. Our experience confirms the fact that adoption of an elegant, declarative style for program development not only makes the code more clear and readable, it also leads to an increase in the amount of parallelism inherent in the application exposed.

We are currently studying the possibility of using parallelism to improve the precision of the results produced by Artwork. For tractability, the rules for resolving temporal ambiguity are distinct from the rules for resolving speech act ambiguity. The temporal rules are applied in a first pass through the data, and the speech act rules are applied to the results of the first pass. This way, Artwork avoids considering all possible combinations of temporal and speech act features. However, the speech-act information cannot assist the system in resolving the temporal ambiguity. With a parallel implementation, more combinations can be processed, and this allows to perform a limited amount of integrated temporal and speech act processing.

References

Adriaens, G., and Hahn, U. 1994. *Parallel Natural Language Processing.* Ablex Publishing.

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

M. Carlson and G. Gupta (eds.) 1996. *Journal of Logic Progr. (Special Issue on High-Performance Implem.)* 29(1-3).

M. Walker and J. Moore (eds.) 1997. *Computational Linguistics (Special Issue on Empirical Studies in Discourse)* 23(1).

DeGroot, D. 1984. Restricted AND-Parallelism. In *Conf. on 5th Generation Computer Systems.*

Devos M. et al. 1988. The Parallel Expert Parser. In *Proceedings of COLING.*

Farwell, D., and Wilks, Y. 1991. Ultra: A multilingual machine translator. In *Mach. Transl. Summit.*

Gupta, G., Pontelli, E. et al. 1994. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, 93–109. MIT Press.

Hermenegildo, M., and Greene, K. 1990. &-Prolog and its Performance. In *ICLP90*, MIT Press.

Hermenegildo, M. et al. 1995. Incremental analysis of logic programs. In *ICLP95.* MIT Press.

Hirschman L. et al. 1988. Or-parallel Speedup in Natural Language Processing. In *ICLP88.* MIT Press.

Levin, L. et al. 1995. Using Context in the Machine Translation of Spoken Language. In *Proc. Theoretical and Methodological Issues in Machine Translation.*

Manousopoulou A. et al. 1997. Automatic Gneration of Portable Parallel Natural Language Parsers. In *Int. Conf. on Tools with AI.* IEEE Computer Society.

Matsumoto, Y. 1986. A Parallel Parsing System for Natural Language Analysis. In *Int. Conf. on Logic Programming.* Springer Verlag.

Pereira, F. and Shieber, S.M. 1987. *Prolog and Natural-Language Analysis.* CSLI Lecture Notes, number 10. Cambridge University Press.

Pereira, F., and Warren, D. 1980. Definite Clause Grammars for Language Analysis. *AI*, 13.

Pontelli, E. 1997. *High-Performance Parallel Execution of Prolog Programs.* Ph.D. Dissertation, NMSU.

Pontelli, E., Gupta, G. et al. 1997. Automatic Compile-time Parallelization of Prolog Programs for DAP. In *ICLP97.* MIT Press.

Reithinger, N., and Maier, E. 1995. Utilizing statistical dialogue act processing in verbmobil. In *Proceedings ACL*, 116–122.

Rosé, C. et al. 1995. Discourse processing of dialogues with multiple threads. In *Proceedings of ACL*, 31–38.

Slator, B. 1988. Lexical Semantics and Preference Semantics Analysis. Technical Report MCCS88143, Computing Research Laboratory.

Trehan, R., and Wilk, P. 1988. A Parallel Chart Parser for the CCND Logic Languages. In *ICLP88.* MIT Press.

Wiebe J. et al. 1996. ARTWORK: Discourse Processing in Machine Translation of Dialog. Technical Report MCCS96294, Computing Research Laboratory.

Wiebe J. et al. 1997. An Empirical Approach to Temporal Reference Resolution. In *Proceedings 2nd Conference on Empirical Methods in NLP.*