Parallelizing Description Logics

Frank W. Bergmann and J. Joachim Quantz
Technische Universität Berlin, Projekt KIT-VM11

Abstract. Description Logics (DL), one of the major paradigms in Knowledge Representation, face efficiency problems due to large-scale applications, expressive dialects, or complete inference algorithms. In this paper we investigate the potential of parallelizing DL algorithms to meet this challenge. Instead of relying on a parallelism inherent in logic programming languages, we propose to exploit the application-specific potentials of DL and to use a more data-oriented parallelization strategy that is also applicable to imperative programming languages. We argue that object-level propagation is the most promising inference component for such a parallelization, as opposed to normalization, comparison, or classification.
We present two alternative PROLOG implementations of parallelized propagation on a loosely coupled MIMD (Multiple Instruction, Multiple Data) system, one based on a *farm* strategy, the other based on *distributed objects*. Whereas the farm strategy yields only poor results, the implementation based on distributed objects achieves a considerable speedup, in particular for large-size applications.

1 Introduction

In the last 15 years Description Logics (DL) have become one of the major paradigms in Knowledge Representation. Combining ideas from Semantic Networks and Frames with the formal rigor of First Order Logic, research in DL has focussed on theoretical foundations [Donini Et Al. 91] as well as on system development [Brachman Et Al. 91] and application in real-world scenarios [Quantz, Schmitz 94].

Whereas in the beginning it was hoped that DL provide representation formalisms which allowed efficient computation, at least three trends in recent years caused efficiency problems for DL systems and applications:

- a trend towards expressive dialects;
- a trend towards complete inference algorithms;
- a trend towards large-scale applications.

With the current state of technology it seems not possible to build a DL system for large-scale applications which offers an expressive dialect with complete inference algorithms. The standard strategy to cope with this dilemma is to restrict either expressivity, or completeness, or application size.

In this paper we investigate an alternative approach, namely a parallelization of Description Logics. Due to physical limitations in performance gains in

conventional processor architectures, parallelization has become more and more important in recent years. This comprises parallel structures inside processors as well as outside by scaling several processors to parallel systems.

Several fields of high-performance computing already adopted to this new world of paradigms, such as image processing [Burkhard Et Al 94], finite element simulation [Diekmann Et Al 94], and fluid dynamics [Strietzel 94]. We expect that in future parallelism will become a standard technique in the construction of complex AI applications.

A standard approach to parallelization in the context of logic programming concentrates on the development of parallel languages that exploit the parallelism *inherent* in the underlying logic formalism ([Clark, Gregory 87], [Pontelli, Gupta 94] and many more). In this paper we will follow a rather different approach which analyzes a particular application, namely Description Logics. The parallelization we propose uses *explicit* parallelism based on the notion of processes and messages that is programming language independent.

In the next section we give a brief introduction into Description Logics and the parallelization potential of DL inference algorithms. In Section 3 we describe two different strategies of parallelizing object-level propagation in DL systems. The corresponding implementations are evaluated in detail in Section 4.

2 Description Logics

In this section we give a brief introduction into Description Logics. In doing so we sketch the main inference components of a DL system and point out their complexity and their potential for parallelization.

Basically, the alphabet of a Description Logic contains *concepts* (unary predicates), *Roles* (binary predicates), and *objects* (individual constants). DL dialects vary wrt the term-forming operators they support, e.g. conjunction, disjunction, and negation for concepts and roles; value restrictions and number restrictions for concepts; composition and inversions of roles.

Three types of formulae are usually distinguished in DL systems, namely *term introductions*[1] ($t_n :< t$ or $t_n := t$), *rules* ($c_1 => c_2$), and *object descriptions* ($o :: c$). Given a modeling, i.e. a list of such formulae, DL systems basically answer two types of queries:

$$\mathrm{t}_1 \; ? < \mathrm{t}_2$$
$$\mathrm{o} \; ? : \mathrm{c}$$

The first query succeeds iff the term t_1 is subsumed by the term t_2(i.e. t_2 is more general than t_1), the second one iff the object o is an instance of the concept c.

Two main reasoning paradigms are used in DL systems. Originally, inferencing was realized by *normalize-compare* algorithms, which first transform concepts and roles into normalforms and then structurally compare these normalforms. Note that the normalforms of objects are similar to the normalforms of concepts and can thus be generated and compared by the same algorithms.

[1] We use 'term' to designate both concepts and roles.

At the end of the 1980's *tableaux methods*, as known from FOL were applied to DL (e.g. [Donini Et Al. 91]). The resulting subsumption algorithms had the advantage of providing an excellent basis for theoretical investigations. Not only was their correctness and completeness easy to prove, they also allowed a systematic study of the decidability and the tractability of different DL dialects.

The main disadvantage of tableaux-based subsumption algorithms is that they are not constructive but rather employ refutation techniques. Thus in order to prove the subsumption $c_2 \sqsubseteq c_1$ it is proven that the term $c_1 \sqcap \neg c_2$ is inconsistent, i.e. that $o :: c_1 \sqcap \neg c_2$ is not satisfiable. Though this is straightforward for computing subsumption, this approach leads to efficiency problems in the context of retrieval. In order to retrieve the instances of a concept 'c' in a situation 's', we would in principle have to check for each object 'o' whether $\Gamma \cup \{o :: c \text{ in } s\}$ is satisfiable.[2]

In most existing systems, on the other hand, inference rules are more seen as production rules, which are used to pre-compute part of the consequences of the initial information. This corresponds more closely to Natural Deduction or Sequent Calculi, two deduction systems also developed in the context of FOL. A third alternative, combining advantages of the normalize-compare approach and tableaux-based methods has therefore been proposed in [Royer, Quantz 92]. The basic idea is to use *Sequent Calculi* instead of tableaux-based methods for the characterization of the deduction rules. Like tableaux methods, sequent calculi provide a sound logical framework, but whereas tableaux-based methods are refutation based, i.e. suitable for theorem checking, sequent calculi are constructive, i.e. suitable for theorem proving.

Based on the ideas presented in [Royer, Quantz 92, Royer, Quantz 94] the DL system FLEX has been developed at the Technische Universität Berlin. The parallelization described in the following has been performed for the FLEX system. The general techniques are applicable to all normalize-compare systems, however, and the following presentation does not rely on the particular features of FLEX.

In the remainder of this section we briefly sketch three main inference components of DL systems, describe their realization within the normalize-compare paradigm, and evaluate their potential for parallelization.

Subsumption Checking. To test subsumption between two terms, both terms are first *normalized* and then structurally compared. The format of normalization rules depends on the expressiveness of the DL dialect. For the purpose of this paper it is sufficient to consider normalforms as sets of atoms and normalization rules as having the form

$$\alpha_1, \ldots, \alpha_n \rightharpoonup \beta$$

i.e. if the atoms $\alpha_1, \ldots, \alpha_n$ are contained in a normalform, then β is added to this normalform.

[2] See [Schaerf 94] for tableaux-based algorithms for object-level reasoning and [Kindermann 95] for a discussion of efficiency problems.

In the comparison phase it is then checked whether for each atom in the subsuming normalform we find an atom in the subsumed normalform which is more specific.

Classification. When processing the term introductions, each term name is classified, i.e. compared with all previously introduced names. As a result *subsumption hierarchies* for concepts and roles are obtained, which are directed acyclic graphs. Classification is basically realized by searching direct supers and direct subs in the subsumption hierarchy, i.e. by a number of subsumption checks between the new term and previously introduced terms.

Propagation. The two reasoning components described so far are usually called *terminological reasoning*. We will now turn our attention towards *assertional reasoning*, i.e. reasoning on the object level. The main difference between terminological and assertional reasoning is that the former is inherently local, whereas the latter is inherently global. In principle we can distinguish between a local phase and a nonlocal phase in object-level reasoning.

In the local phase we determine for an object the *most specific concept* it instantiates. This can be done by using the standard normalize and compare predicates and the search for direct supers in the classification component. Thus we normalize the description of an object thereby obtaining a normal form and compare it with the normal forms of the concepts in the hierarchy. In addition to this standard classification we also have to apply DL rules when processing objects. This is achieved by applying all rules whose left-hand sides subsume the object's normal form. After this application the normal form is again normalized and classified until no new rules are applicable [Owsnicki-Klewe 88].

In the nonlocal phase we have to propagate information to other objects. For illustration consider the following propagation rules:

$$o_1 :: \mathrm{all}(r,c)\ \&\ r{:}o_2 \rightharpoonup o_2 :: c$$
$$o_1 :: r{:}o_2 \&\ \mathrm{atmost}(1,r),\ o_2 :: c \rightharpoonup o_1 :: \mathrm{all}(r,c)$$
$$o_1 :: r_1{:}o_2,\ o_2 :: r_2{:}o_3 \rightharpoonup o_1 :: r_1 \mathrm{comp}\ r_2{:}o_3$$
$$o_1 :: r{:}o_2 \rightharpoonup o_2 :: \mathrm{inv}(r){:}o_1$$

We call these rules nonlocal since information at an object o_1 can have impact on an object o_2. Depending on the "connectivity" of the objects, adding a new description at an object can thus cause a reclassification of arbitrarily many other objects.

Parallelization. In principle, all three inference components show some parallel potential. We will argue, however, that parallelizing propagation is the most promising. The reason for this is that the basic operations in normalization, comparison, and classification are rather fine-grain, compared with the message passing overhead of MIMD systems.

Object-level propagation, on the other hand, is an ideal candidate for our parallelization strategy. Each propagation is rather time-consuming and causes additional propagations which can be straightforwardly parallelized since they

are both independent and monotonic. In the following section we will present two different strategies for parallelizing propagation.

3 Parallelization Strategies

We begin by noting several relevant properties of object-level propagation. As already indicated above propagation of information from one object to another can cause additional propagation to other objects. This kind of 'ping-pong' interaction terminates only when a 'fixed point' is reached and no new information is produced. Since propagation in Description Logics is monotonic, we can execute propagations in an arbitrary order, always ending up with the same result. We will refer to this property as *confluence*.

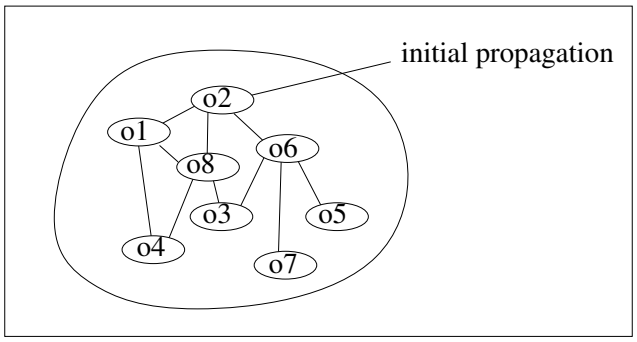


Fig. 1. A group of objects interchanging propagations.

FLEX Data Flow. Figure 3 shows the first few stages after the start of a propagation process. In this example every propagation causes three other propagations. This creates a 'chain reaction', thus increasing the number of 'pending propagations' exponentially. This rise will stop as soon as the new information (from the object tell) becomes more and more integrated into the network.

This results in a smaller 'fan out' that leads to a decrease of pending propagations until the fixed point is reached. Figure 3 shows qualitatively the increase and decrease of pending propagations with respect to to propagation steps. Please note that the steps in the middle part take much longer than the steps at the sides, because each processor (only a limited number of processors available) has to compute several propagations.

Given the analysis of the FLEX data flow, we consider two parallel paradigms as potential candidates for an implementation: The *farm* paradigm and the *distributed objects* paradigm. In the remainder of this section we briefly present these two alternatives. Theoretical considerations and numerical calculations towards efficiency and scalability can be found in the detailed analysis in [Bergmann 95].

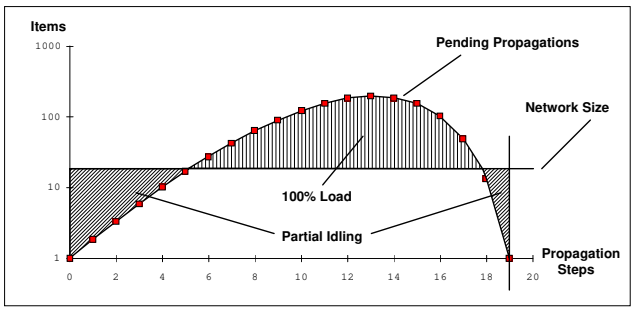


Fig. 2. Exponential increase of propagations.

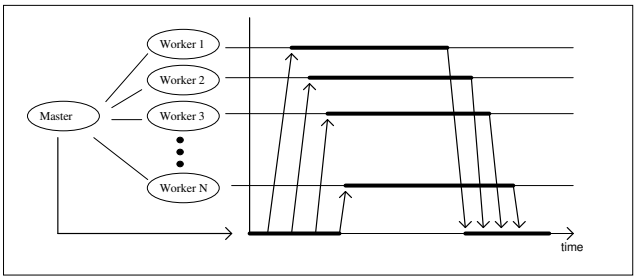


Fig. 3. Timing of the farm communication scheme.

Farm Parallelism. The *farm* communication structure shown in Figure 3 is widely used in industrial applications such as image processing [Burkhard Et Al 94] and finite element simulation [Diekmann Et Al 94]. It is theoretically well known and there exists a variety of strategies to distribute workload evenly across a network.

A farm consists of several parallel processes with a fixed structure: one process is called 'master' and is responsible to distribute tasks to a group of 'worker' processes which perform their tasks in parallel and return control and results back to the master. Farm structures are frequently used to parallelize applications that can be split into subtasks with a priori known duration. Examples are image processing or finite element systems. From a theoretical point of view, there are two potential sources of inefficiency in this architecture:

1. uneven distribution of workload and
2. a communication bottleneck created by the centralized position of the master.

Communicating Objects Parallelism. In the *communicating-objects* paradigm the central management institution (master) of the farm parallelism is replaced by

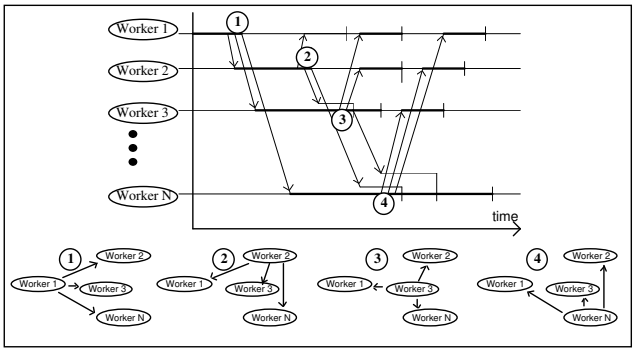


Fig. 4. Communication events and workload distribution during the first two Propagation Stages.

(local) knowledge of all objects about the 'addresses' of their neighbors. Objects communicate directly with each other, in contrast to the centered communication scheme of the farm. This helps to avoid communication bottlenecks in a network. The general difference between a farm and a network of communicating objects is the different perspective of parallelism: Within a farm, tasks are distributed; within the distributed objects scheme, objects are distributed.

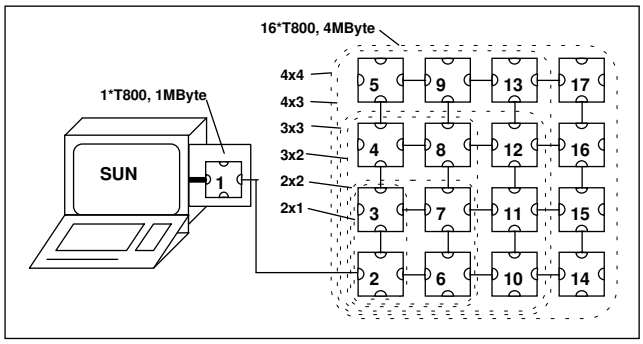
This approach appears to be similar to the agent-based paradigm developed by distributed AI research [Dossier 91]. In contrast to this approach, objects within FLEX have to be considered elements of a distribution strategy rather then independently interacting entities. With respect to the definition given in [Bond, Gasser 88] we have to subsume our efforts here under the field of 'distributed problem solving'.

For an effective balancing of workload, certain assumptions about tasks and the computational environment have to be made. In our case, all processors can be assumed to behave identical and the statistical distribution of the task length is assumed to be narrow. Uneven distributions of workload can finally be treated by special load balancing algorithms (see below).

4 Experimental Results

We chose the 'Parsytec Multicluster II' machine as base for the parallel implementation of FLEX. It consists of 16 processing nodes that each contain an INMOS T800 Transputer with 4 MByte of RAM. Each Transputer consists of a RISC processing kernel, a memory interface and 4 DMA driven serial interfaces, called 'links'. Each link has a transfer rate of approximately 1.2 MByte/s and all 4 links can run independently together with the RISK kernel, hardly affecting processing performance (communication delays could be neglected). This hardware platform is especially suitable to serve as a testbed for parallel

implementations due to its simple architecture and the availability of comfortable development environments. However, it does not provide state of the art computational performance and suffers substantially from memory restrictions.



Topology	2x1	2p1	2x2	3x2	3x3	4x3	4x4
Processors	3	4	5	7	10	13	17
Workers	1	2	3	5	8	11	15

Fig. 5. Hardware configuration and working nodes.

Figure 5 shows the topologies used for the tests in this chapter and the number of available worker nodes. The overhead of 2 processors is due to memory limitations. Processor 1 could not be used because its 1 MByte RAM is not sufficient to hold the FLEX code. Processor 2 is used to hold the 'shell' process that synchronizes the generation of new objects. Normally this process can be located somewhere in the network and would not consume any computing performance, but in this case it had to be separated due to memory restrictions.

The language used to implement Distributed FLEX is a Prolog dialect called Brain Aid Prolog (BAP). It represents a 'standard' Prolog with parallel library extensions, implementing a scheme similar to CSP [Hoare 85]. Parallelism and synchronization is expressed explicitly using the notion of 'processes' and 'messages'. A process in BAP is a single and independent Prolog instance with a private database. A message is any Prolog term that becomes exchanged between two processes. Messages are send and received using the send_msg(Dest, Msg) and rec_msg(Sender, Msg) predicates. Message sender and destination are identified by their 'process id' (PID). Messages are routed transparently through the network. The order of messages is maintained when several messages are send from the same sender to the same destination. When a message reaches its destination process, it is stored in a special database, called 'mailbox'. Each process owns its private mailbox in which the messages are stored FIFO.

Although the development of parallel FLEX was greatly simplified by the way BAP expresses parallelism, it is possible to apply the same parallel techniques

to future FLEX implementation in programming languages such as LISP or C.

The main area of FLEX applications within the KIT research group is Natural Language Processing (NLP) [Quantz, Schmitz 94]. Unfortunately memory limitations kept us from using these applications as benchmarks. Instead we imitate the structure of our NL applications leading to benchmarks with similar behavior but much lower memory requirements.

Base of our considerations is the fact that in the NL applications each propagation creates a certain number of propagations to other objects. This results in an 'avalanche' of propagations, rising exponentially until the systems slowly reaches a fixed point. The average fan out (the number of propagations following an initial propagation) is a measure to describe this avalanche effect and turned out to be a major factor in the system performance.

r :< rtop	o1 :: r:o3 and r:o2 and r:o8
c1 :< all(r,c2)	o2 :: r:o4 and r:o7 and r:o2
c2 :< all(r,c3)	o3 :: r:o7 and r:o2 and r:o1
c3 :< all(r,c1)	o4 :: r:o1 and r:o8 and r:o6
	o5 :: r:o2 and r:o7 and r:o8
	o6 :: r:o1 and r:o7 and r:o5
	o7 :: r:o3 and r:o8 and r:o4
	o8 :: r:o7 and r:o4 and r:o6
	o1 :: c1

Fig. 6. A sample benchmark with 8 Objects, 3 Concepts and Fanout 3.

To evaluate the FLEX performance with benchmarks of different sizes, we created a benchmark generator that is capable of generating randomly connected networks of objects. These benchmarks maintain a structure similar to our application while consuming much less memory resources. [3]

	(seq)	1	2	3	5	8
c10_3_2	(58)	78	63	55	56	58
c20_3_2	(177)	253	185	159	160	162

Fig. 7. Execution times (seconds) for the farm parallelization.

Figure 7 shows the execution times for the farm benchmarks. The first row contains the benchmark names that are composed by three numbers that indicate

[3] [Bergmann 95] analyzes quantitatively the influence of the avalanche *exponent* on the applicability of parallel execution.

the number of objects, concepts and fanout respectively (for example 'c20_5_3' means that the benchmark consists of 20 objects, 5 concepts and a fan out of 3). The following rows give the execution times with respect to the number of processors. The '(seq)' fields gives the reference time of the (original) sequential version of FLEX.

The parallelization of FLEX using the farm paradigm showed very poor results. This can be explained by the rather high costs to distribute the system state out to the workers and to integrate the computation results back into the system state. Both activities have to be done sequentially, thus slowing down the parallel execution.

Although there is some potential for optimizing the FARM implementation, we stopped the development and focused on the distributed-object version of FLEX.

	1	2	3	5	8	11	15
c10_3_2	(1.0) 59	(1.9) 30	(1.8) 32	(3.3) 18	(3.3) 18	(2.8) 21	(3.3) 18
c10_3_3	(1.0) 43	(1.5) 28	(1.5) 28	(2.4) 18	(2.7) 16	(2.5) 17	(2.9) 15
c10_3_4	(1.0) 327	(2.0) 159	(2.3) 141	(3.1) 105	(3.8) 87	(4.4) 74	(4.4) 73
c20_3_2	(1.0) 179	(1.8) 97	(3.0) 59	(3.1) 58	(4.0) 45	(4.8) 37	(4.5) 40
c20_3_3		145	129	58	56	66	51
c20_3_4		240	173	164	70	74	68
c20_5_3		527	355	173	155	141	160
c20_5_4		344	453	411	185	126	189
c40_3_2		314	176	137	105	111	72
c40_3_3			258	231	141	111	87
c40_3_4			569	467	264	319	230
c80_3_2			1032	665	225	200	181
c80_3_3					443	336	266
c80_3_4					947	662	583

Fig. 8. Benchmark Execution Times.

The parallelization of FLEX using the distributed objects paradigm turned out to be a lot more promising. Figure 8 shows the absolute execution times of the considered benchmarks. The names of the benchmarks are composed as in Figure 7.

Note that the execution times in Figure 8 are measured with an accuracy of ± 2 seconds. The sequential execution times (entries in the '1' row) for several benchmarks are not available due to the memory limitations. This means that it is not possible to calculate the relative speedup in such a line (future tests on Transputer machines with more memory will fill these gaps). This is the reason why we omited the speedup figures in all but the first 4 benchmarks.

The table shows high speedups (efficiencies $> 80\%$) for all benchmarks, if

the number of objects exceeds the number of processors by a certain factor (between 5 and 10). This result can be interpreted by the perspective of Section 3, where we saw that network efficiently is dependent on the number of pending propagations in the network. If this number is too low, few processing nodes are busy, resulting in a bad network efficiency.

Within [Bergmann 95] the quantitative analysis shows that the propagation-processor ratio is more relevant to system performance than the overhead caused by message passing. [4] It also indicates how these problems can be overcome, allowing for even larger networks.

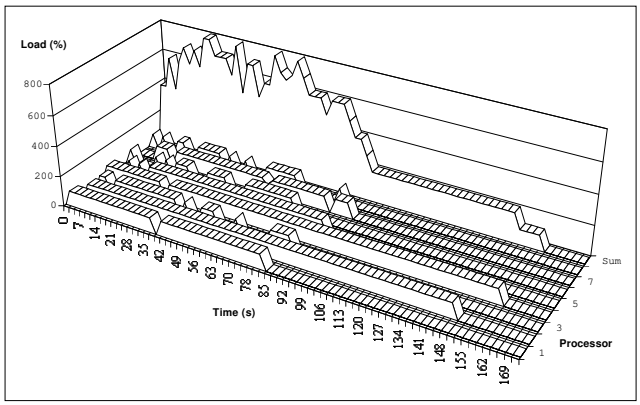


Fig. 9. Runtime behavior of distributed FLEX within a 3x3 Network

A major problem for all distributed systems is the balance of load inside the network. Within distributed FLEX each object represents a potential load. Unfortunately the presence of objects is only a statistical measure for load, while the actual distribution depends on runtime conditions. The illustration in Figure 9 depicts execution of a benchmark with an uneven load distribution. The Transputers 2 and 4 slow down the overall performance. It is easy to see that the network is quite busy during the first half of the execution time (ca. 75% efficiency). At the second half, all object servers have terminated, except two (ca. 25% efficiency). This leads to a reduction of the overall efficiency to ca. 50% and explains the variation of the results in Figure 8

The necessary optimization of the uneven distribution of processor load over the network can be achieved by temporarily 'relocating' objects to other processors. Such a mechanism would be capable of reducing overhead time created by loads remaining on a few processors. We are currently implementing this optimization.

[4] This is valid for Transputer systems with 2..256 processors, 2D matrix topology and shortest path routing algorithm

5 Conclusion

The results of the parallel implementation of FLEX are in general very satisfying. We achieved high speedups with benchmarks that are structurally similar to the real-world applications in natural language processing ($> 80\%$ for benchmarks that fit the size of the network). The efficiency of execution rises mainly with the *propagation/processor* ratio and thus with the application size. This is an important result because especially large applications are to be considered candidates for a parallel implementation. Theoretical considerations [Bergmann 95] show that there are only few technical limits to the scalability of the distributed objects implementation.

We have to state that the Transputer system under consideration here is not applicable to real world problems due to its poor overall performance and its memory restrictions. Ideal candidates for such implementations are parallel computers with large (local) memory resources and high communication bandwidth. Alternatively, shared-memory multiprocessor workstations fulfill all requirements for an efficient parallelization.

We assume that the communication structure of FLEX is similar to many other applications in Artificial Intelligence. In particular, applications involving complex, forward-chaining inferencing are potential candidates for a parallelization based on the distributed-objects approach presented in this paper.

References

[BAP 93] F.W. Bergmann, M. Ostermann, G. von Walter, "Brain Aid Prolog Language Reference" Brain Aid Systems, 1993

[Bergmann 95] F.W. Bergmann, *Parallelizing FLEX*, KIT Report in preparation, TU Berlin

[Bond, Gasser 88] A. Bond, L. Gasser, "Readings in Distributed Artificial Intelligence", Morgan Kaufmann, Los Angeles, CA, 1988

[Brachman Et Al. 91] R. Brachman, D.L. McGuiness, P.F. Patel-Schneider, L. Alperin Resnick, A. Borgida, "Living with CLASSIC: When and How to Use a KLONE-like Language", in J. Sowa (Ed.), *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, San Mateo: Morgan Kaufmann, 1991, 401–456

[Burkhard Et Al 94] H. Burkhard, A. Bienick, R. Klaus, M. Nlle, G. Schreiber, H. Schulz-Mirbach, "The Parallel Image Processing Sytem PIPS" in R. Flieger, R. Grebe (eds), *Parallelrechner Grundlagen und Anwendung* IOS Press, Amsterdam, Netherlands, 1994, 288–293

[Clark, Gregory 87] K. Clark, S, Gregory, "PARLOG: Parallel Programming in Logic" in E. Shapiro (ed), *Concurrent Prolog* The MIT Press, Cambridge, Massachusetts, 1987, 84 – 139

[Diekmann Et Al 94] R. Diekmann, D. Meyer, B. Monien, "Parallele Partitionierung unstrukturierter Finite Elemente Netze auf Transputernetzwerken" in R. Flieger, R. Grebe (eds), *Parallelrechner Grundlagen und Anwendung* IOS Press, Amsterdam, Netherlands, 1994, 317–326

[Donini Et Al. 91] F.M. Donini, M. Lenzerini, D. Nardi, W. Nutt, "The Complexity of Concept Languages", KR'91, 151–162

[Dossier 91] A.C. Dossier, "Intelligence Artificielle Distribuee", *Bulletin de l'AFIA*, **6**, 1991

[Hoare 85] C. A. R. Hoare, "Communicating Sequential Processes" Prentice Hall, Englewood Cliffs, N.J., USA, 1985

[Kindermann 95] C. Kindermann, *Verwaltung assertorischer Inferenzen in terminologischen Wissensbanksystemen*, PhD Thesis (submitted), TU Berlin, 1995

[Owsnicki-Klewe 88] B. Owsnicki-Klewe, "Configuration as a Consistency Maintenance Task", in W. Hoeppner (Ed.), *Proceedings of GWAI'88*, Berlin: Springer, 1988, 77–87

[Pontelli, Gupta 94] E. Pontelli, G. Gupta, "Design and Implementation of Parallel Logic Programming Systems", *Proceedings of ILPS'94 Post Converence Workshop* 1994

[Quantz, Schmitz 94] J.J. Quantz, B. Schmitz, "Knowledge-Based Disambiguation for Machine Translation", *Minds and Machines* **4**, 39–57, 1994

[Royer, Quantz 92] V. Royer, J.J. Quantz, "Deriving Inference Rules for Terminological Logics", in D. Pearce, G. Wagner (eds), *Logics in AI, Proceedings of JELIA'92*, Berlin: Springer, 1992, 84–105

[Royer, Quantz 94] V. Royer, J.J. Quantz, "On Intuitionistic Query Answering in Description Bases", in A. Bundy (Ed.), *CADE-94*, Berlin: Springer, 1994, 326–340

[Schaerf 94] A. Schaerf, *Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues*, Dissertation Thesis, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", 1994

[Strietzel 94] "Large Eddy Simulation turbulenter Strömungen auf MIMD-Systemen" in R. Flieger, R. Grebe (eds), *Parallelrechner Grundlagen und Anwendung*, IOS Press, Amsterdam, Netherlands, 1994, 357 - 366

This article was processed using the LaTeX macro package with LLNCS style