EXPLOITING PARALLELISM IN UNIFICATION-BASED PARSING

M. P. van Lohuizen
Dept. of Information Technology and Systems - PDS
Delft University of Technology
Delft, The Netherlands
mpvl@acm.org

Abstract
Because of the nature of the parsing problem, unification-based parsers are hard to parallelize. We present a parallelization technique designed to cope with these difficulties.

Parallel parsing of natural language has been researched extensively. In [6] we can find an overview of parallel chart parsing. Most attempts, however, were not very successful [8, 3]. Only recently two NLP applications were successfully parallelized [7, 5]. However, the former focussed on Prolog and the latter exploits coarse-grained parallelism of the kind that proved unusable for our Deltra system[1] or other systems [3]. We present a more widely applicable approach, not limited to Prolog.

Most unification-based parsers have characteristics that make them particularly hard to parallelize. Typically, unifications account for the bulk of the processing time in unification-based parsing. However, parallelizing this operation is difficult [1] and does not speed up the CF part. Therefore, most research has focussed on exploiting parallelism at the CF level, where the unification operations are atomic and distributed amongst processors. This approach has several problems as well. First, each item has a different impact on the derivation of new items. In addition, the computational cost of unifying the items in one parse can vary wildly. These irregularities make it hard to find a good distribution of work (*load balancing*). Another characteristic is the lack of *locality*, causing excessive communication and ineffective cache utilization. Finally, because all intermediate results need to be recorded in a chart, there is a lot of *synchronization* between processors. This aspect is aggravated by the need of dynamic load balancing, which requires additional synchronization.

The implementation is aimed at *shared-memory* architectures, mainly driven by the difficulties discussed before. The increasing availability of these systems further justifies this choice. Let us first consider the problem of load balancing. Experiments in [10] indicate that only dynamic load balancing of single unification operations can yield a scalable solution to parallel parsing. However, putting each individual unification in a central queue will result in too much overhead. We solve this dilemma by taking a *work stealing* approach:[2] each processor—or thread[3]—has its own work queue. However, whenever a processor runs out of work, it may steal work from other processors. This considerably decreases the amount of synchronization because now synchronization is only required when a processor is stealing work. We further limit the number of steals by allowing a thread to steal an amount of work proportional to the amount of work available at the victim.

[1]The Deltra system comprises a medium sized grammar and dictionary for Dutch.
[2]The work-stealing approach is similar to the Cilk-5 system [2], but differs in its optimization for chart parsers.
[3]There is one thread for each processor. Threads are automatically distributed amongst processors by the OS.

As mentioned before, synchronization is also required to store results in the chart. Letting threads wait unconditionally for exclusive access incurs too much contention. We solve this problem by splitting all tasks in distributed and sequential parts. Instead of contending for chart access, a thread simply queues the sequential work[4] and proceeds with other work. The scheduler [9] ensures that at most one processor is processing sequential work, automatically providing synchronization.

A final optimization improves on the locality. Our parser allows synchronization of the chart per grammar rule. By associating each grammar rule—and its data—with a single processor, we can improve caching behavior. This does not compromise on the ability to balance load, because assigned work may still be stolen. In general, the scheduler has been designed to move most overhead to the stealing side, leaving minimal overhead for normal operation.

Experiments confirmed that work stealing indeed yields a well-balanced distribution of work. On a 4-thread setup, 6% of the total time was consumed by idling and 6% could be attributed to scheduling overhead.[5] Work stealing proved to be necessary, as omitting it yielded 20% idling for a 2-thread setup (rapidly increasing for more processors). The solution to reduce synchronization proved effective as well. Only 1/2000 to 1/5000 synchronizations per unification were required on a 2-processor run. The ratio of running time of the sequential version and the 1-processor parallel version $T_1//_sT$ was 1.03, yielding minimal overhead. Running a 2-processor version on a dual Pentium-II yielded a speedup of 1.7 (1.4 for a Pentium-Pro). Experiments verified that the gap in speedup between this result and the load balancing simulation was caused by communication of the (hardware) cache coherency protocol. This indicates that the bottleneck is caused by ineffective cache usage. Currently our research is focusing on improving cache utilization to mitigate the usage of memory bandwidth.

In conclusion, the speedup that can be obtained depends on the specific grammar being used. Nevertheless, the presented scheduling method allows for the finest possible grain of parallelism, without resorting to parallel unification. In addition, although originally designed for the parallelization of chart parsers, the presented technique can be applied to any tabulation programming technique.

References

[1] C. Dwork et. al.. On the sequential nature of unification. *J. of Logic Programming*, 1(1):35–50, 1984.

[2] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, May 1998.

[3] G. Görz, et. al. Research on architectures for integrated speech/language systems in Verbmobil. In *The 16th COLING*, vol. 1, pp. 484–489, Copenhagen, DK, Aug 5–9 1996.

[4] H.J. Honig. A new double dotted parsing algorithm. TR 94-108, Delft University of Technology, 1994.

[5] A.G. Manousopoulou et. al. Automatic generation of portable parallel natural language parsers. In *Proc. of the 9th Conf. on Tools with Artificial Intelligence (ICTAI '97)*, pp. 174–177. IEEE CS Press, 1997.

[6] A. Nijholt. Overview of parallel parsing strategies. In M. Tomita, ed. *Current Issues in Parsing Technology.* Kluwer Academic Publishers, Norwell, MA, 1991.

[7] E. Pontelli, G. Gupta, J. Wiebe, and D. Farwell. Natural language multiprocessing: A case study. In *Proc. of the 15th Nat. Conf. on Artificial Intelligence (AAAI '98)*, July 1998.

[8] H.S. Thompson. Parallel parsers for context-free grammars–two actual implementations compared. In G. Adriaens et. al., ed. *Parallel Natural Language Processing.* Ablex Publ. Corp., Norwood, NJ, 1994.

[9] M.P. van Lohuizen. Parallel processing of natural language parsers. In *PARCO '99*, 1999.

[10] M.P. van Lohuizen. Simulating communication of parallel unification-based parsers. PDS Report Series PDS-1998-008, Delft University of Technology, 1998.

[4]We use a queuing mechanism that does not requires locking.

[5]This result was obtained from a multi-threaded run on 1 processor to abstract from cache coherency effects.