PARALLELIZATION OF INCREMENTAL PATTERN MATCHING IN GRAPH TRANSFORMATION

Gábor BERGMANN
Advisor: Dániel VARRÓ

I. Introduction

Nowadays, desktop computers are often equipped with multi-core processors, and single-threaded execution does not take advantage of this increased computational capacity. It is a great challange in the industry to find algorithms that are scalable and capable of exploiting the power of modern computing architectures. Model transformation is an application domain that could benefit greatly from parallelizations (along with other improvements to efficiency), as processing large-scale models often suffer from permormance issues.

Using a graph transformation [1] based approach for model transformations, there are even more possibilities for the exploitation of parallelism. Besides model manipulation sequences, graph transformations involve a graph searching phase, which is targeted at finding the matches of a graph pattern. Nevertheless, graph transformation tools rarely exploit parallel execution.

Previous work revolved aroung improving pattern matching performance by employing an incremental strategy [2] based on the RETE [3] algorithm. Incremental techniques store the occurrence set of graph patterns so that they are always immediately available, and update these caches upon model changes. RETE in particular stores the match set of subpatterns also. A RETE net consists of cache nodes, each responsible for maintaining the match set of a subpattern, and update channels between these nodes. Changes in the model or subpattern caches trigger messages flowing in these channels to prompt the incremental update of the cache stored at the recipient node.

This paper examines ways in which RETE-based pattern matching could benefit from parallelism.

II. Concurrent pattern matching and model manipulation

Contrary to previous work, the RETE net implementation used throughout this paper relies on *asynchronous* message passing. Using asynchronous messaging, the load on the main thread of the transformation can be reduced by executing the incremental pattern matcher (which consumes change messages from the queue) in a separate thread. When the transformation manipulates the model (see Fig. 1), it only has to send the new update message to the message queue, and continue its operation. The thread of the pattern matcher will execute the update propagation in the background, ideally, without imposing a performance penalty on the transformation thread. When the message queue becomes empty, the RETE network has reached steady state; the pattern matcher thread then goes to sleep and will not resume its operation until a new update message is posted.

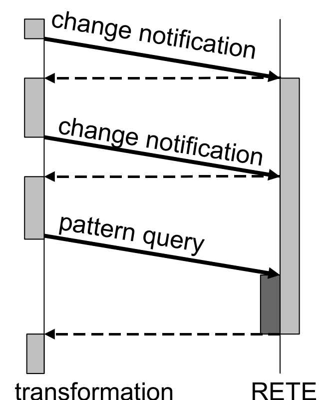

Figure 1: Separate pattern matcher thread with concurrency and waiting

When the transformation initiates pattern matching, it has to assure that background update propagations have terminated and the matches stored at the production nodes are up-to-date. If the network has not yet reached its fixpoint, the model manipulation thread will have to sleep until that happens.

Performance expectations. While the local search based pattern matchers operate with cheap model changes and costly pattern queries, a sequential RETE-based matcher [2] relies upon a moderate overhead on model change balanced by instant pattern queries. This novel concurrent incremental pattern matching approach combines the advantages of the former two: it has cheap model manipulation costs, and potentially instant pattern queries. Although the transformation might have to wait for the termination of the background pattern matcher thread, the worst case of this time loss is still comparable with the update overhead of the original RETE approach.

III. Multi-threaded pattern matching with RETE

The concurrent pattern matching approach can be improved further by parallelizing the update propagation phase. Here I present a simple solution. The basic idea is to employ multiple pattern matcher threads to consume update messages. The proposal splits the network into separate RETE *container*s, each of which is responsible for matching a set of subpatterns. Each container has its own distinct set of nodes, and a dedicated pattern matcher thread consuming update messages of a dedicated queue. Each container is responsible for forwarding messages to its nodes using the dedicated message queue. Forwarding messages between two containers is accomplished by enqueueing the message in the target container. Fig. 2 depicts a multi-threaded pattern matcher illustrating how a RETE net can be split into several containers for parallel execution.

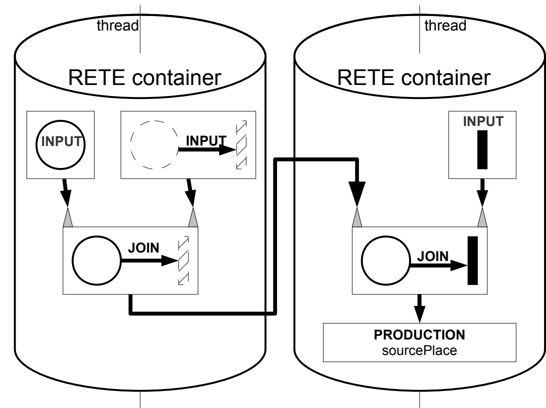

Figure 2: Multiple containers

Performance expectations. An ideal application scenario would be several parallel transformations that are known to use different patterns; allowing straightforward splitting and parallelization of the RETE net, with a low amount of inter-connectedness. By partitioning the patterns into relatively independent containers, a multi-threaded RETE pattern matcher may achieve high performance.

IV. Conclusion

I have designed ways of parallelizing a RETE-based incremental graph pattern matcher and implemented them as part of the VIATRA2 framework [4]. This approach supports large-scale model transformation and complements the parallelization of transformation execution. Future work is required to carefully measure performance in various problem classes, and to fine-tune the implementation.

References

[1] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools, World Scientific, 1999.

[2] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró, "Incremental pattern matching in the VIATRA model transformation system," in *Graph and Model Transformation (GraMoT 2008)*, G. Karsai and G. Taentzer, Eds. ACM, 2008.

[3] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, 19(1):17–37, September 1982.

[4] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Sci. Comput. Program.*, 68(3):214–234, 2007.