Implementing Graph Transformations in the Bulk Synchronous Parallel Model

Christian Krause[1], Matthias Tichy[2], and Holger Giese[3]

[1] SAP Innovation Center, Potsdam, Germany, christian.krause01@sap.com
[2] Chalmers | University of Gothenburg, Sweden, matthias.tichy@cse.gu.se
[3] Hasso Plattner Institute, University of Potsdam, holger.giese@hpi.uni-potsdam.de

Abstract. Big data becomes a challenge in more and more domains. In many areas, such as in social networks, the entities of interest have relational references to each other and thereby form large-scale graphs (in the order of billions of vertices). At the same time, querying and updating these data structures is a key requirement. Complex queries and updates demand expressive high-level languages which can still be efficiently executed on these large-scale graphs. In this paper, we use the well-studied concepts of graph transformation rules and units as a high-level modeling language with declarative and operational features for transforming graph structures. In order to apply them to large-scale graphs, we introduce an approach to distribute and parallelize graph transformations by mapping them to the Bulk Synchronous Parallel (BSP) model. Our tool support builds on Henshin as modeling tool and consists of a code generator for the BSP framework Apache Giraph. We evaluated the approach with the IMDB movie database and a computation cluster with up to 48 processing nodes with 8 cores each.

1 Introduction

Graph-based modeling and analysis becomes relevant in an increasing number of domains including traditional business applications such as supply chain management and product lifecycle management, but also in non-traditional application areas such as social network analysis and context-aware search [1]. In many of these areas, there is a trend towards collecting more data with the effect that the big-data dimension of the graph processing problem becomes a limiting factor for existing modeling and analysis approaches. On the one hand, there is a demand for high-level, declarative modeling languages that abstract from the basic underlying graph operations such as traversals and node and edge manipulations. On the other, these high-level, declarative modeling languages must be executed efficiently also on large-scale graphs (in the size of billions of vertices and edges).

In the last decades, the theory of graph transformations (see, e.g., [2]) evolved to a very active field both on the foundational and the application side. Graph transformations provide high-level modeling concepts for graph processing with both declarative and operational parts. However, most of their today's applications are, e.g., in model management, model transformation and software architectures where only recently issues with big models have been started to get addressed (e.g., [3]). Because of this and the fact that graph transformations are algorithmically challenging, little effort has been taken so far to make the concepts usable also for big data problems. Specifically, the recent work on large-scale graph processing focuses on relatively simple graph queries for vertex-labeled graphs [4,5,6,3]. While these approaches support distribution of large-scale graphs on several compute nodes in a cluster and parallelized execution of queries, the expressive power of these approaches is very limited (see the discussion of related work in Section 2). On the other hand, distribution of graphs in the area of graph transformations is currently only considered for modeling purposes [7], but not for physically distributing large-scale graphs on several machines. Thus, the high expressive power of graph transformations can currently not be used to solve big data problems which rely on truely parallel and distributed graph processing.

To make the high-level modeling concepts of graph transformations available for processing of large-scale graphs, we map the concepts of (declarative) transformation rules and (operational) transformation units [8] to the bridging model *Bulk Synchronous Parallel* (BSP) [9] which provides an abstraction layer for implementing parallel algorithms on distributed data. Thereby, we enable the use of the expressive language concepts of graph transformations for solving big data problems. In our prototypical tool suppport we use the Henshin [10] graph transformation language and tool to specify transformation rules and units. We have implemented a code generator which takes Henshin models as input and generates code for the BSP-based framework Apache Giraph [11]. Giraph builds on the infrastructure of Apache Hadoop [12] which implements the MapReduce [13] programming model. Our choice for BSP and Giraph was driven by fact that they provide the required support and infrastructure for transparent distribution and parallelization including load-balancing. We use a synthetic and a real-data example to show the feasability and the scalability (horizontal and vertical) of our approach. We also define modeling concepts tailored for parallel graph processing.

Organization The rest of this paper is organized as follows. In Section 2 we compare our approach to existing work in this area. In Section 3 we give an overview of the relevant background. Section 4 describes our mapping of graph transformation to the BSP model and run-time optimizations. Section 5 contains an experimental evaluation. Section 6 discusses conclusions and future work.

2 Related Work

An algorithm for subgraph matching on large-scale graphs deployed on a distributed memory store is introduced in [4]. The approach is limited to vertex-labeled graphs without attributes and edge labels. More advanced matching conditions such as negative application conditions are not supported and transformations are also not considered. These restrictions apply also to the work on distributed graph pattern matching in [5] and [6]. Moreover, only graph simulations (as opposed to subgraph-isomorphy checking) are considered which are less powerful but can be checked in quadratic time.

The distributed graph transformation approach developed by Taentzer [7] focuses on the modeling of distributed graphs and transformations, but not on physical distribution on several machines. This is also the case for the parallel graph transformation approach for modeling Timed Transition Petri Nets in [14]. Parallelizing (incremental) graph pattern matching and graph transformations is discussed in [15]. Again, the approach does not consider distributing a large-scale graph on a cluster of compute nodes and therefore also does not support horizontal scalability. Furthermore, all matches are stored in memory and thus the approach does not scale for large graphs with a high number of matches due to the memory restrictions. Large-scale graph processing based on a mapping to an in-memory relational database is discussed in [1]. Scalability is not investigated here. Early results on distributed and massive parallel execution of graph transformations on similar sized graphs are shown in [3]. However, it is unclear which type of matching conditions are supported and how the approach scales for rules with a high number of partial matches as in our example.

Blom et al. present a distributed state space generation approach for graph transformations [16] based on LTSmin [17]. In contrast to our work, the framework does not allow to distribute the graph over a cluster. Instead the clients store complete states and send newly created states to itself or other clients for further subsequent generation. This approach is not applicable to large-scale graphs targeted in our work. State space generation for graphs of this size is neither our target nor reasonable.

Besides Giraph [11], Pregel [18] is another implementation of the Bulk Synchronous Parallel model on graph data. An architectural difference is that Giraph builds upon the MapReduce [13] framework Apache Hadoop [12] which is widely used and available.

3 Background

3.1 Graph Transformations with Transformation Units

Our modeling concepts build on the theory of algebraic graph transformations for typed, attributed graphs [2]. Specifically, we consider directed graphs with vertex and edge types, and primitive-typed vertex attributes. Transformations for these graphs are defined using declarative transformation rules and procedural transformation units [8,10].

In our approach, we consider transformation rules as graphs extended with stereotypes for vertices and edges, and conditions and calculations on attribute values. We use the following vertex and edge stereotypes: ⟪preserve⟫, ⟪delete⟫, ⟪create⟫, ⟪require⟫ and ⟪forbid⟫. Applying a rule consists of finding a match of the rule in the host graph and performing the operations indicated by the stereotypes. The stereotypes ⟪require⟫ and ⟪forbid⟫ have special meanings and are used for defining positive and negative application conditions (PACs and NACs), respectively. For attributes, we use expressions to constrain the allowed values and to calculate new values.

Transformation units are a means to define control-flows for transformations. We consider here a subset of the transformation unit types supported by the Henshin [10] model transformation tool. An *iterated unit* executes another unit or rule a fixed number of times. A *loop unit* executes a rule as long as possible, i.e., until no match is found. A *sequential unit* executes a list of units or rules in a fixed order. An *independent unit* nondeterministically selects one rule or unit from a set of rules or units and executes it.

Parallel Execution Semantics

In this paper, all rule applications are maximum parallel, i.e., rules are by default applied to *all* found matches in the host graph. The rationale behind this is to enforce parallelism

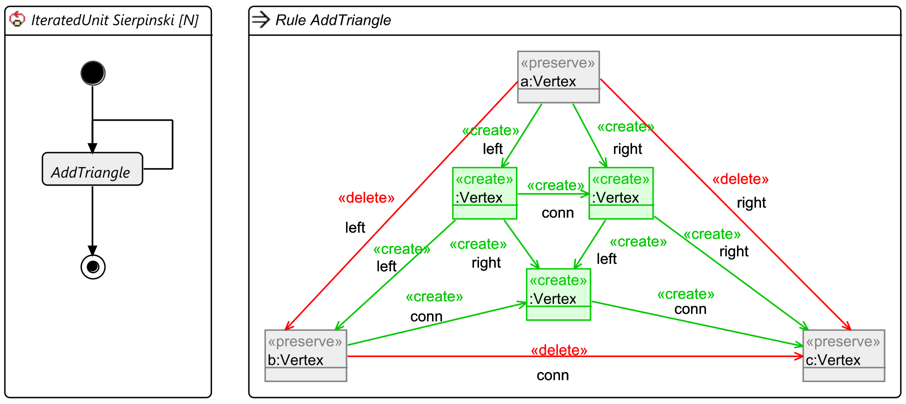

Fig. 1. Iterated unit and rule for constructing Sierpinski triangles of depth N.

already at the modeling level. This is necessary in order to actually parallelize the execution of the transformations at run-time. Also, we assume that for any two overlapping matches during the parallel application, the graph modifications are conflict-free, e.g., if a vertex is preserved in one match, it cannot be deleted in another match. Different rules are applied in the order as specified in Henshin, e.g., sequentially in a sequential unit. Regarding transformation units, we note that we do not consider any transactional behavior, i.e., there is no rollback on unsuccessful execution of subunits.

Example 1 (Sierpinski triangle). Fig. 1 shows an example of a transformation unit and a rule for constructing Sierpinski triangles modeled using Henshin. The transformation unit on the left is an iterated unit which executes the rule *AddTriangle* on the right N times. Since in our semantics in every iteration the rule is applied in parallel to all matches, executing the transformation unit on a 3-vertex triangle generates the full Sierpinski triangle of depth N.

3.2 Bulk Synchronous Parallel (BSP) on Graphs

Bulk Synchronous Parallel (BSP) is a bridging model for implementing parallel algorithms which was developed by Leslie Valiant in the 1980s [9]. Nowadays, BSP is a popular approach for efficiently processing large-scale graph data. Implementations of BSP for graph data include Apache Giraph [11] which is used at Facebook to analyze social networks, and Pregel [18], a graph processing architecture developed at Google. In both of these frameworks BSP is used as computational model for implementing highly parallel algorithms on very large graphs distributed in a cluster, that supports both horizontal and vertical scaling.[4] Standard examples of BSP applications are computing shortest paths and the well-known PageRank algorithm which is used to rank websites in search results.

[4] Horizontal scaling (*scale-out*) refers to adding nodes to a compute cluster. Vertical scaling (*scale-up*) refers to adding resources, e.g., main memory or processing cores, to single nodes.

Algorithms following the BSP bridging model must adhere to a specific scheme. In particular, any algorithm implemented in BSP is executed as a series of *supersteps*. In a superstep, every vertex in the graph can be either active or inactive. A superstep constitutes a unit of computation and interaction consisting of four components:

1. **Master computation:** Single computation executed centrally on a master node, mainly used for bookkeeping and orchestrating the vertex computations.
2. **Vertex computation:** Concurrent computation executed locally for every active vertex of the graph. This part can be highly parallelized.
3. **Communication:** During the vertex computation, vertices can send messages to other vertices, which will be available in the next superstep.
4. **Barrier synchronization:** Before the next superstep is started, the vertex computation and communication of the current superstep must be finished for all vertices.

The master computation can be seen as an initialization phase for every supersteps. In the parallel vertex computations, incoming messages are processed, local computations for the current vertex are performed, messages can be sent to other vertices, and the vertex can be requested to become inactive. Inactive vertices do not take part in the vertex computations of the next supersteps. However, an inactive vertex becomes active again if it receives a message. In a vertex computation, messages can be either send to adjacent vertices or vertices with known IDs. For instance, a received message can contain such a vertex ID. The final step of every superstep is a barrier synchronization. Specifically, the next superstep can be started only when all vertex computations and communications of the current superstep are finished. The BSP algorithm ends when all vertices are inactive.

In addition to the computation and communication, vertices can *mutate*, i.e. change, the graph during the vertex computation. It is important to note that –analogously to the communication– the effects of graph mutations are visible only in the next superstep.

Example 2 (BSP computation). Fig. 2 illustrates the run of an example BSP algorithm. In fact, is shows already an application of the graph transformation *AddTriangle* in Example 1 realized as a BSP computation. The run consists in total of 5 supersteps. In superstep I, vertex 0 sends the message [0] to vertex 1. In supersteps II and III this message is extended and first forwarded to vertex 2 and then back to vertex 0. Note that in I and II the messages are sent via the *l*- and *r*-edges, respectively, whereas in III vertex 2 extracts the vertex ID from the message in order to send it back to vertex 0. In superstep IV, no more messages are sent. Instead, a number of graph mutations are performed, specifically: 3 edges are removed, 3 new vertices and 9 new edges are created. The details of the mapping from transformations to BSP are explained in the Section 4.

In addition to the inter-vertex communication, vertices can also send data to dedicated, centrally managed *aggregators*. Aggregators process all received data items using an associative, commutative aggregation operation to produce a single value available to all vertices in the next superstep. We distinguish between *regular* and *persistent* aggregators, where the former are being reset in every superstep and the latter not. Using the sum of integers as aggregator operations, it is for instance possible to compute the total number of exchanged messages – either only for the last superstep (using a regular aggregator) or the total number for all previous supersteps (using a persistent aggregator).

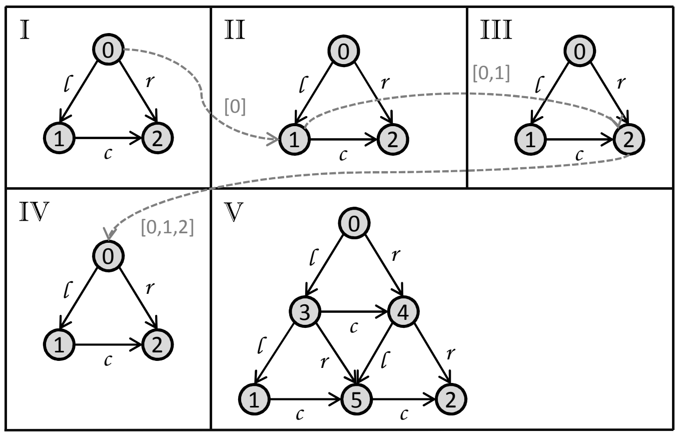

Fig. 2. Illustration of a BSP computation that realizes an application of the *AddTriangle* rule.

4 Implementing Graph Transformations in BSP

Example 2 indicates already that the BSP model is generic and powerful enough to realize graph transformations with it. In this section, we show that this is in fact the case for arbitrary complex transformation rules and control-flows modeled using transformation units. We identified three main challenges in implementing graph transformations in BSP: 1) to realize the rule pattern matching with only local knowledge, 2) to coordinate the distributed and parallel execution of rules and transformation units, and 3) to deal with the complexity of the graph pattern matching problem. In addition, we propose the new concept of *attribute aggregators* to benefit from the parallelism not only in the execution phase, but also in the modeling. Finally, we discuss run-time optimizations of the graph pattern matching.

4.1 Graph Pattern Matching and Rule Applications

To apply a rule requires to find all its matches in the distributed graph using a parallel algorithm that is designed based on the BSP-principles. To this end, we propose to split up the matching process into a series of *local steps*[5]. In each local step, local constraints are checked and sets of partial matches are generated, extended or merged. After all constraints have been checked and the final set of matches has been computed, the rule can be applied by performing the required graph modifications in parallel.

The most performance-critical part of a rule is the pattern matching. For a given rule, we statically construct a search plan which defines the operations for every local step in the matching phase. A simplified version of the search plan generation algorithm is shown in Listing 1.1. For simplicity, we consider here only connected graph patterns with at least one edge. The algorithm generates a search plan in the form of a list of

[5] Local steps are relative to a specific rule; supersteps are defined for a complete transformation.

Listing 1.1. Simplified algorithm for search plan generation of connected graph patterns.

```
/* Function for generating a search plan for a graph pattern.
 * Input 'pattern': a connected graph pattern to be searched.
 * Output: search plan as list of edges.
 */
generateSearchPlan(pattern) {
  visited := $\emptyset$     // set of visited vertices
  result := []     // search plan as list of edges
  while (|visited| < |pattern.vertices|) {
    traversals := $\emptyset$     // set of traversals as set of list of edges
    for (start $\in$ pattern.vertices)
      if (start $\notin$ visited) {
        t := dfsTraversal(start, visited)     // depth−first traversal stopping at visited vertices
        traversals := traversals $\cup$ { t }
      }
    next := sup(traversals)     // select longest traversal
    result := result++ next     // append to search plan
    visited := visited $\cup$ next.vertices
  }
  return result
}
```

edges along which the matches are constructed. The algorithm iteratively builds maximal, ordered subgraphs of the pattern until the complete pattern is covered. The generated search plan starts with a depth-first traversal of the largest traversable subgraph of the pattern. The search plan continues with depth-first traversals of the remaining parts of the pattern, until the search plan contains the complete pattern graph. For example, the generated search plan for the *AddTriangle* rule in Fig. 1 is given by the following list of edges: $[(a -left\rightarrow b), (b -conn\rightarrow c), (a -right\rightarrow c)]$.

The list of edges generated by the search plan function is translated into a series of local steps. In such a local step, local constraints of a vertex, such as type information, attribute values, existence of edges and injectivity of matches are checked. During the pattern matching, a set of local matches is maintained and step-wise extended. The partial matches are forwarded as messages to either adjacent vertices (for extending a partial match by new bindings of nodes) or vertices that have been matched already (for checking an edge between bound objects or merging partial matches). When the complete search plan has been processed, the final set of matches is used to perform the rule modifications in parallel. Note that both the pattern matching as well as the graph modifications are executed in parallel in this model. A simplified version of the generated matching code for the *AddTriangle* rule is shown in Listing 1.2.

Graph pattern elements with the stereotype ⟪require⟫ take a special role in our parallel graph transformation approach. The difference to the ⟪preserve⟫ stereotype is that the existence of the elements is checked, but they are not considered as part of the match. This is a useful modeling feature to avoid overlapping and conflicting matches. We give a specific example for this modeling concept in Section 5.

Listing 1.2. Simplified generated matching code for the *AddTriangle* rule.

```
/*
 * Generated matching function for the AddTriangle rule.
 * Input 'vertex': active vertex in the host graph.
 * Input 'matches': list of matches received in the current superstep.
 * Input 'step': currently executed local step.
 */
matchAddTriangle(vertex, matches, step) {
  targets := ∅
  switch (step) {
   0: if (vertex.value = TYPE_VERTEX ∧ |vertex.edges| ≥ 2) {      // matching vertex "a"
         match := [vertex.id]
         for (edge ∈ vertex.edges)
           if (edge.value = TYPE_VERTEX_LEFT ∧ edge.target ∉ targets) {
             sendMessage(edge.target, match)
             targets = targets ∪ {edge.target}
           }
       }
       break
   1: if (vertex.value = TYPE_VERTEX ∧ |vertex.edges| ≥ 1)      // matching vertex "b"
         for (match ∈ matches) {
           if (vertex.id ∈ match) continue      // injectivity check
           match := match ++ [vertex.id]
           for (edge ∈ vertex.edges)
             if (edge.value = TYPE_VERTEX_CONN ∧ edge.target ∉ targets) {
               sendMessage(edge.target, match
               targets = targets ∪ edge.target
             }
         }
       break
   2: if (vertex.value = TYPE_VERTEX)      // matching vertex "c"
         for (match ∈ matches) {
           if (vertex.id ∈ match) continue      // injectivity check
           match := match ++ [vertex.id]
           sendMessage(match[0], match)
         }
       break
   3: for (match ∈ matches)      // checking for "right" edge
         for (edge : vertex.edges)
           if (edge.value = TYPE_VERTEX_RIGHT ∧ edge.target = match[2]) {
             applySierpinski(match)      // apply the rule w.r.t. the found match
             break
           }
       break
  }
}
```

4.2 Transformation Units

Transformation units provide control-flow constructs to coordinate the execution of rules. In our BSP-based approach, transformation units are managed during the master computation. The master computation maintains a unit execution stack in the form of a persistent aggregator. The elements on this stack are pairs of unit or rule IDs and local step indizes. The unit ID is used to decide which unit or rule is active, and the local step determines the current execution phase in this unit or rule. In a rule, the local step defines the current stage in the matching phase. In a sequential unit, the local step is used to store the index of the currently executed subunit, and similarly for other unit types. In addition to the unit execution stack, we maintain a rule application counter in the form of a regular aggregator. It stores the number of rule applications in the last superstep and is required to decide when loop units should be terminated.

For an example of generated code for a transformation unit, we refer to the online resources provided for the example in Section 5.2.

4.3 Attribute Aggregators

In many graph transformation approaches, attribute values of vertices can be set during a rule application using an expression that takes as parameters other attribute values of matched vertices. These expressions are usually limited to the scope of a single match. In our approach, however, rule applications are always maximum parallel, i.e., always applied to all found matches. Since during a rule application all matches are readily available, we can define attribute calculations that are not limited to the scope of a single match, but a set of matches, potentially all of them.

To facilitate global attribute calculations, we introduce the concept of *attribute aggregators*, which are associative and commutative operations on attribute values (similarly to aggregation functions in relational databases). Specifically, we consider the following set of pre-defined attribute aggregators: COUNT, MIN, MAX, SUM and AVG which respectively count occurences, compute the minimum, the maximum, the sum and the average of numerical attribute values. We distinguish between local and global attribute aggregators. Global attribute aggregators use the attribute values of all found matches. In local attribute aggregators, if the aggregator is used in an attribute calculation of a vertex v in the rule and v is matched to a vertex x, then all matches where v is matched to x are used. We give an example of a local attribute aggregator in Section 5.2.

4.4 Run-time Optimizations

The performance of the BSP-based graph transformations mainly depends on the efficiency of the match finding. The most effective way to improve it is to reduce the number of partial matches generated during the matching. We realized two optimizations.

To reduce the number of symmetric partial matches, we introduced a static analysis that finds symmetries in PACs. During the matching phase, symmetric matches to the PACs are automatically discarded. As a second approach to reduce the number of partial matches during the matching, we consider *segmentation*. The idea is to partition the vertices of the host graph into a set of disjoint segments. Each segment is individually

Number of workers	2	4	6	8	10	12
Execution time (seconds) for levels 1–15	281.8	195.7	166.3	133.7	114.9	112.6
Execution time (seconds) for level 16	559.1	376.8	295.6	197.9	172.1	158.2
Execution time (seconds) for level 17	–	–	896.4	635.0	526.3	489.7

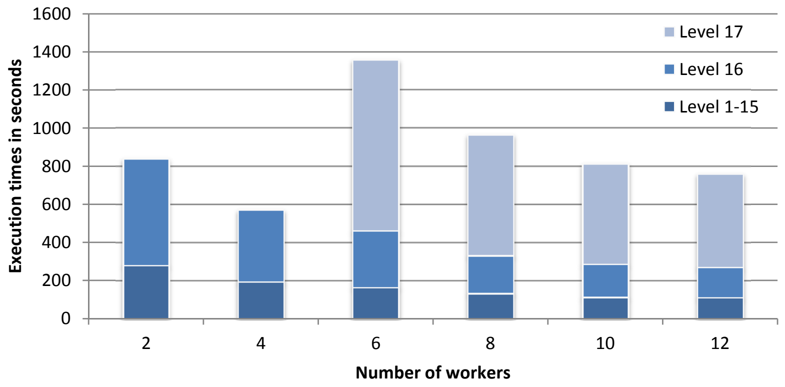

Fig. 3. Execution times of the iterated unit *Sierpinski* for different numbers of workers.

examined during the matching phase. Specifically, the matching process starts with vertices from one of these segments and continues until all segments were used. Matches from previous segments are kept during the matching. Thus, we partially sequentialize the matching process to reduce the memory consumption for partial matches.

5 Evaluation

We used the Henshin [10] tool to model transformation rules and units and implemented a code generator that takes Henshin models as input and produces Java code for the BSP-framework Apache Giraph [11]. Except for the parallelism, the Giraph-based semantics of rules is the same as in the Henshin interpreter. We tested this using a test suite that currently consists of 15 non-trivial example transformations. We conducted our experiments on a small cluster consisting of 6 slave nodes, each of them with 120GB main memory and Intel Xeon® CPU with 24 cores at 2.30GHz, connected via InfiniBand. To investigate the horizontal and the vertical scalability, we varied the number of Giraph workers between 2 and 12 where we used a maximum number of 6 compute threads per worker. The speed improvements up to 6 workers are primarily horizontal scaling, whereas the speed improvements between 6 and 12 workers are vertical scaling effects.

5.1 Synthetic Example: Sierpinski Triangles

We use the iterated unit in Fig. 1 to construct the Sierpinski triangle of depth N. Note that the size of the result graph is exponential in N. Using our set-up of 6 slave nodes, we built the Sierpinski triangle of depth $N{=}17$ which consists of ≈194 million vertices and 387 million edges. One parallel rule application of the *AddTriangle* rule requires 4 local steps, totaling in 68 supersteps for the whole transformation for $N{=}17$, which required 12 minutes. Fig. 3 shows a comparison of the aggregated run-times of the transformation for different number of workers. The difference from 2 to 6 workers is a horizontal scaling effect, which is a speed-up of factor 1.9. The difference from 6 to 12 workers is a vertical scaling effect, which is an additional speed-up of factor 2. Note that for <6 workers we were only able to compute up to N=16 due to insufficient memory.

We also ran this example in the non-distributed, non-parallel interpreter of Henshin. The Henshin solution was still 25% faster than the 12-worker version of Giraph. We believe that this is due to the additional communication and coordination overhead of Giraph. Due to the memory limitations of one machine, the Henshin solution worked only up to N=16. Note that due to the exponential growth, the graph sizes and number of matches for N=17 is 3 times, and for N=18 already 9 times larger than for N=16.

5.2 Real-Data Example: Movie Database

As a real-data example, we used the IMDB movie database[6] dated July, 26th 2013. The database contains 924, 054 movies (we consider only movies and not TV series), 1, 777, 656 actors, and 980, 396 actresses. We use the simplified metamodel / typegraph shown in Fig. 4, which contains classes for movies, actors and actresses. Additionally, we introduce the new class Couple, which references two persons. The goal of this transformation is that for every pair of actors / actresses which played together in at least three movies, we create a Couple object. This new couple object should contain references to all movies that the couple occurred in. Moreover, the couple has a numerical attribute to store the average rank of all movies the couple appeared in.

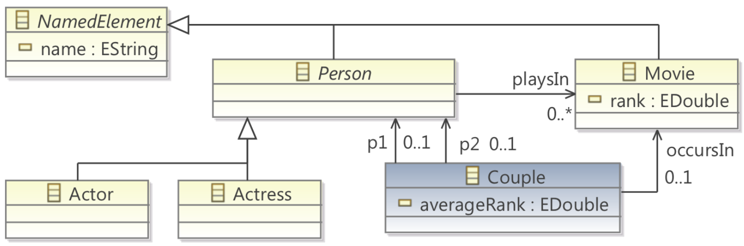

Fig. 4. Simplified and extended metamodel for IMDB movie dataset.

We use the transformation rules and the sequential unit shown in Fig. 5 to solve this task. First, the rule *CreateCouple* generates Couple vertices for every pair of persons that played together in at least three movies. It is important to ensure that only one Couple vertex is created for every such pair. Thus, the movie vertices are matched as a PAC (using ⟪require⟫ stereotypes), i.e., their existence is checked but they are not considered as part of the matches. To avoid symmetric couples, we use an additional attribute condition that enforces a lexicographical order of the names of the two persons.

[6] Obtained from `http://www.imdb.com/interfaces`.

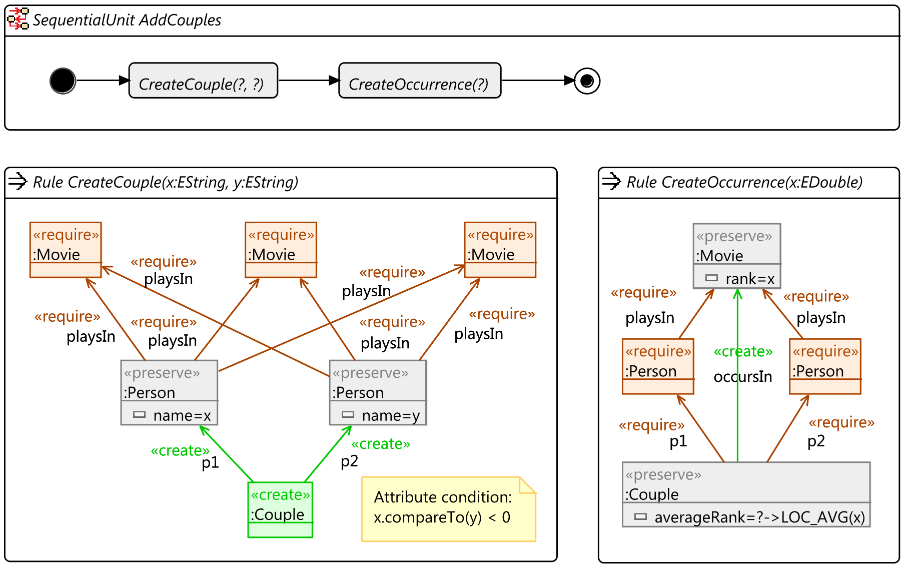

Fig. 5. Sequential unit and rules for adding couple nodes.

In the second part, the rule *CreateOccurrence* is used to add links between the newly created couple vertices and the movies they played in. Moreover, we use the local attribute aggregator LOC_AVG (see Section 4.3) to compute the average rank of all movies the couple occurred in. The expression averageRank=?→LOC_AVG(x) denotes an assignment of the attribute averageRank with the new value LOC_AVG(x). Note that since both rules are each applied in parallel to all possible matches, our solution to this example does not require any use of loops and can thus be fully parallelized.

We highlight here the main challenges in this example. First, the data set is too large (in the order of millions of vertices) to solve the task using brute force, i.e., by naively enumerating all possible solutions and checking the graph and attribute constraints of the rules. Second, the fact that navigation is possible only from persons to movies, but not vice versa (which is an instance of a performance bad smell as described in [19]), makes it difficult to reduce the possible matches for the persons. Third, the matching of the three movies is highly symmetrical and can cause an unnecessary high number of partial matches during the matching.

We generated code for this example transformation.[7] In this example, we benefit from the optimizations described in Section 4.4. This reduced the number of symmetric matches of the PACs by a factor of 6. However, the number of partial matches during the matching was still too high. Therefore, we also used segmentation with 100 segments.

We executed the transformation on our 6-slave node cluster. The graph before the transformation had 3,635,741 vertices and 5,615,552 edges, and afterwards 5,328,961

[7] Models and generated source code available at `http://www.eclipse.org/henshin/examples.php?example=giraph-movies`.

Number of workers	2	4	6	8	10	12
Execution time (seconds) for 60% graph size	939	327	162	172	125	113
Execution time (seconds) for 80% graph size	2,273	757	376	378	295	253
Execution time (seconds) for 100% graph size	5,254	1,499	762	751	561	479

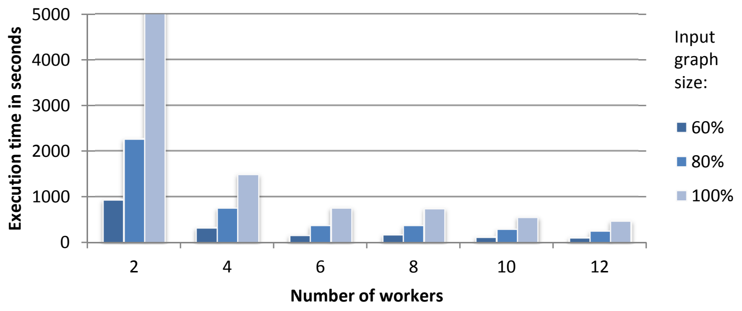

Fig. 6. Execution times of the couples example (2-movie version) in seconds.

vertices and 16,933,630 edges.[8] Because of the relatively high number of segments, the execution time was approx. 6 hours. In addition, we ran a variant of this transformation with only two common movies per couple where we did not require segmentation. Fig. 6 shows the execution times for this version. The overall performance improved by a factor of 11 when switching from 2 to 12 workers.

In addition to the small 6-node cluster, we ran this example (with two common movies) on a part of the Glenn cluster at Chalmers. In this set-up, each node uses 2 AMD Opteron® 6220 processors (8 cores each at 3GHz) with 32GB of main memory. For this benchmark, we varied the number of nodes between 5 and 24. For each node, we used 3 Giraph workers with 6 threads each. The memory was restricted for each Giraph worker to 10GB. Fig. 7 shows the resulting execution time averaged over 3 runs. On the left y-axis of the plot, the execution time per node is shown whereas on the right y-axis of the plot, the total execution time is shown. The latter is the product of the execution time and the number of nodes used in order to see how much total time of the cluster is used. Both figures show a good horizontal scalability until 14 nodes are used. Please note that during each evaluation run, the rest of cluster was also used by other jobs and thus could influence the results by network traffic. This is a possible explanation for the spike in the execution time for 20 nodes. More nodes still benefit the execution, but due to increasing fixed costs and communication overhead, the performance does not increase at the same rate.

Finally, we used the standard Henshin interpreter (version 0.9.10) to compute all matches of the *CreateCouple* rule on a single machine using dynamic EMF models. Henshin was not able to compute a single match for the same movie database as used in the other evaluations. This is because nested conditions, such as PACs and NACs, are checked only after a match of the left-hand side of the rule has been found. Specifically,

[8] We validated our transformation 1) using a 25-vertices test input graph, 2) by taking samples from the full output graph, and 3) using similar rules in our test suite.

Number of nodes	5	6	8	10	12	14	16	18	20	22	24
Execution time in seconds	1,220	866	666	587	518	453	430	401	440	387	373
Total node time in seconds	6,104	5,199	5,333	5,879	6,216	6,351	6,880	7,227	8,803	8,529	8,957

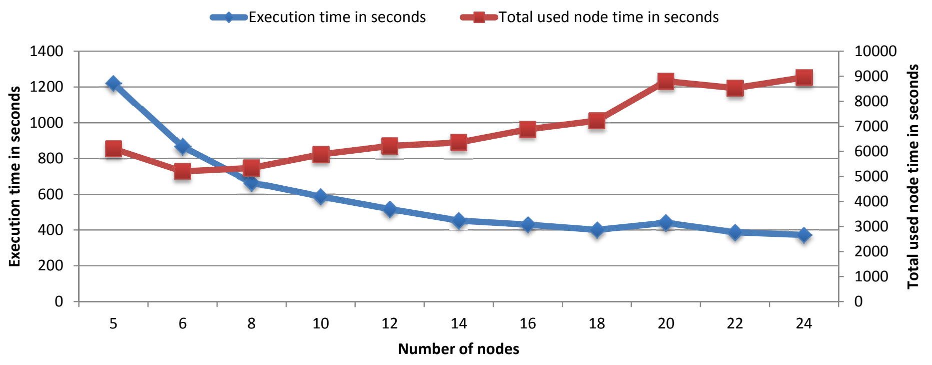

Fig. 7. Execution times of couples example (2-movie version) on the Glenn cluster at Chalmers.

Henshin first fixes matches for the two Person vertices, and then checks the PACs. This matching approach boils down to a brute-force, which is only working for toy input graphs. In our experiments, also alternative models, such as with bidirectional references, nested rules or loop units, were not able to solve the task due to similar problems. Please note that this is mostly an inefficiency of the matching strategy of (this version of) the Henshin interpreter.

6 Conclusions and Future Work

Processing large-scale graph data becomes an important task in an increasing number of application domains. On the one hand, there is a demand for high-level, declarative query and transformation languages for graph data. On the other, the efficient processing of the large-scale graphs is a limiting factor for existing approaches. In this paper, we introduced an approach for the specification of graph querying and processing using graph transformation rules and units which provide a high-level modeling approach with declarative and operational parts. In order to efficiently apply them to large-scale graphs, we mapped graph transformation rules and units to the Bulk Synchronous Parallel model which allowed us to distribute the graphs and to parallelize their processing. We showed using a synthetic and a real-data example that the approach 1) provides new and high-level modeling features that are currently not available for large-scale graph processing, and 2) that the BSP solution provides both horizontal and vertical scalability for efficiently executing these models on large data graphs.

As future work, we plan to investigate more on new modeling features for parallel graph transformations and to optimize the execution in BSP. Concrete next steps involve more fine-tuning in the search plans and a dynamic segmentation approach which chooses the number of segments on-the-fly based on the current workload. In a different line of research we plan to define automatic ways to detect and resolve conflicting parallel matches. Both run-time checks and static analysis, e.g., using a variation of critical pair analysis seem to be relevant here. Finally, we plan to perform more benchmarks, e.g., using the graph transformation benchmark suite described in [20].

References

1. Rudolf, M., Paradies, M., Bornhövd, C., Lehner, W.: The graph story of the SAP HANA database. In: BTW 2013. Volume 214 of LNI., GI (2013) 403–420
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
3. Izsó, B., Szárnyas, G., Ráth, I., Varró, D.: Incquery-d: Incremental graph search in the cloud. In: Proc. of BigMDE '13, ACM (2013) DOI: 10.1145/2487766.2487772.
4. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. Proc. VLDB Endow. **5**(9) (2012) 788–799
5. Ma, S., Cao, Y., Huai, J., Wo, T.: Distributed graph pattern matching. In: Proc. WWW 2012, ACM (2012) 949–958 DOI: 10.1145/2187836.2187963.
6. Fard, A., Abdolrashidi, A., Ramaswamy, L., Miller, J.A.: Towards efficient query processing on massive time-evolving graphs. In: Proc. CollaborateCom 2012, IEEE (2012) 567–574
7. Taentzer, G.: Distributed graphs and graph transformation. Applied Categorical Structures **7**(4) (1999) 431–462 DOI: 10.1023/A:1008683005045.
8. Kreowski, H.J., Kuske, S.: Graph transformation units and modules. Handbook of Graph Grammars and Computing by Graph Transformation **2** (1999) 607–638
9. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8) (August 1990) 103–111 DOI: 10.1145/79173.79181.
10. Arendt, T., Bierman, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. MoDELS 2010. LNCS 6394, Springer (2010) 121–135 DOI: 10.1007/978-3-642-16145-2_9.
11. Apache Software Foundation: Apache Giraph. http://giraph.apache.org.
12. Apache Software Foundation: Apache Hadoop. http://hadoop.apache.org.
13. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1) (January 2008) 107–113 DOI: 10.1145/1327452.1327492.
14. de Lara, J., Ermel, C., Taentzer, G., Ehrig, K.: Parallel graph transformation for model simulation applied to timed transition Petri nets. ENTCS **109**(0) (2004) 17–29 Proc. GT-VMT 2004. DOI: 10.1016/j.entcs.2004.02.053.
15. Bergmann, G., Ráth, I., Varró, D.: Parallelization of graph transformation based on incremental pattern matching. ECEASST **18** (2009)
16. Blom, S., Kant, G., Rensink, A.: Distributed graph-based state space generation. ECEASST **32** (2010)
17. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Proc. CAV 2010. LNCS 6174, Springer (2010) 354–359 DOI: 10.1007/978-3-642-14295-6_31.
18. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proc. SIGMOD 2010, ACM (2010) 135–146 DOI: 10.1145/1807167.1807184.
19. Tichy, M., Krause, C., Liebel, G.: Detecting performance bad smells for Henshin model transformations. In: Proc. AMT 2013, CEUR-WS.org (2013)
20. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: VL/HCC, IEEE Computer Society (2005) 79–88 DOI: 10.1109/VLHCC.2005.23.