Parallel Execution of Model Management Programs*

Sina Madani

Department of Computer Science, University of York, UK
sm1748@york.ac.uk

Abstract. Scalability in Model-Driven Engineering (MDE) remains an open challenge and an active research topic. This paper presents an overview of ongoing work, which aims to significantly improve the performance of model management tasks (e.g., transformation and validation) through parallelisation and distribution. We believe our solution will be highly scalable due to hardware trends; namely, higher core counts, diminishing single-threaded performance improvements and increasing prevalence of cloud computing. We intend to implement our solutions in the Epsilon suite of model management tools and evaluate both the performance gains and correctness of our concurrent implementation(s) using a range of models and test scripts. Initial experiments with parallelisation of the Epsilon Validation Language show promising results both in terms of correctness and performance.

Keywords: Model-Driven Engineering, Concurrency, Parallelism.

1 Introduction and Problem

Scalability is one of the most oft-mentioned challenges associated with Model-Driven Engineering (MDE) [1]. This is especially problematic given that MDE is best suited to large industrial contexts such as automotive (AUTOSAR) and construction (BIM); where models may contain millions of elements. Unfortunately, MDE tools were not initially designed to deal with models of such sizes and therefore present a barrier to adoption [1].

The problem of scalability is a multi-faceted concern. On one front, the issues are in the storage and loading of very large models (VLMs). The current de-facto standard model persistence format (XMI) and modelling framework (EMF) do not support concurrent modification of models, which makes collaboration more difficult. Another front is model management – that is, performing tasks using models – which is typically done using domain-specific languages (DSLs). Such tasks include querying, validation, merging, comparison, transformation and generation of textual artefacts. Although there are several tools which exist for each of these tasks, few (if any) were initially designed for execution over

* This research is in its initial stages (less than 6 months), so technical developments are in their infancy; although the overall research direction and goals are unlikely to undergo major changes.

very large models. This is especially problematic for large models that undergo frequent changes and need re-validating and transforming into other models and/or artefacts, such as real-time data from sensors [2].

This research aims to improve scalability by drastically reducing the execution time of model management programs using concurrency. The scope of this research extends to most model management tasks (i.e., not just model-to-model transformations). For our implementation, we are targeting the Epsilon suite of tools and domain-specific languages.

Epsilon[1] offers multiple DSLs for model management tasks. The rationale for choosing Epsilon is that our research group is familiar with the codebase, although the findings should generalise to other implementations and execution engines. It also allows us to minimise duplication since the DSLs are built on top of a common model-oriented language. Furthermore, it provides an opportunity to investigate the similarities of model management tasks with regards to concurrency and highlight any challenges specific to particular tasks.

2 Related Work

The main computational approaches for improving the performance of model management programs are Incremental, Lazy and Parallel execution.

Incremental Approaches. Incrementality is a technique for avoiding unnecessary re-computations; typically implemented using caching. In a modelling context, the idea is that if only a subset of a source model changes, then the program (e.g., transformation or constraint validation) can be re-executed on the changed subset only. Incrementality is the most commonly explored solution in the MDE literature. Works featuring incremental model-to-model transformations are abundant – for example, in [3]. VIATRA[2] provides incremental querying of EMF models. For model validation, Cabot and Teniente (2006) [4] present a conceptual algorithm which provides the least-work expression to validate a constraint for a given CRUD event. As for model-to-text transformation, the work of Ogunyomi (2016) [5] uses runtime analysis (property access traces) to identify the impact of model changes on the generated output. An interesting tangent to incremental model transformation is proposed in [6], where the statically determinable parts of the program are computed and cached (or inlined directly in the code) prior to execution.

Lazy Approaches. Lazy evaluation refers to the notion of delaying execution until the result is required. For instance, if a program makes a query for some data or a computation but then never uses it, then the computation and/or loading could have been avoided. Tisi et al. (2011) [7] modified ATL with lazy execution semantics for both source consumption and target navigation. Tisi et al. (2015) [8] also added lazy evaluation to OCL collections (iterators) without breaking specification compatibility.

[1] `http://www.eclipse.org/epsilon/` [2] `https://eclipse.org/viatra/`

Parallel and Distributed Approaches. Parallelism is the idea of dividing the computation or data between multiple threads, executing them simultaneously and merging the results. Amongst the most pertinent works to our research is LinTra [9]; which uses the Linda co-ordination language for concurrent execution of model transformations in a distributed setting. Tisi et al. (2013) [10] developed a task-parallel version of the ATL engine by exploiting some desirable properties of its declarative semantics; achieving 2.5x speedup for 1 million elements on a 4 core / 8 thread CPU. Benelallam et al. (2015) [11] developed ATL-MR; a modified ATL engine which uses the popular MapReduce [12] programming model for distributing model transformations. This data-parallel approach achieved 3x speedup on 8 nodes; though efficient load balancing was not considered. In [13], Benelallam et al. (2016) propose an efficient partitioning method based on footprints to compute dependencies on-the-fly.

Gap Analysis. All three approaches to improving performance of programs identified in the literature (laziness, incrementality and parallelism) are orthogonal. For example, laziness and incrementality are often necessary for *reactive* approaches, such as [17]. Incrementality exploits a space-time tradeoff; exchanging increased memory consumption for reduced computation time in the form of caching. However, it provides no performance benefits when the program is executed on the model for the first time or in a different environment, and is less useful if a large proportion of the model is updated.

As far as we are aware, there are currently no model management language execution engines (even in prototype stage) which are reactive (i.e., incremental + lazy) *and* parallel. Furthermore, we are also not aware of any concurrent execution engines which are capable of scaling across both local and distributed computing resources. Although there has been increasing interest in parallel and distributed model transformations in recent years, the same is not true of other model management tasks such as validation, comparison and code generation.

We are also not aware of any works which attempt to execute model management programs on Graphics Processing Units (GPUs). Given the rapid pace of developments in GPU architectures (with no sign of a slowdown) and relatively stagnant CPU improvements in recent years, the potential performance gains from GPU computing could be several orders of magnitude greater than what would be achievable with CPUs. Modern graphics cards are capable of simultaneously executing hundreds of thousands of threads, with enthusiast-grade GPUs boasting over 11 TFLOPs compute performance. Furthermore, modern graphics cards have several gigabytes of high-speed memory, which could be used to store even the largest of models.

3 Proposed Solution

Our proposed solution is to architect and implement data-parallel execution for model management programs. This can potentially reduce execution times to a fraction of the original without reliance on pre-computation (as with incremental

approaches). This is further justified by hardware developments; as almost all modern general-purpose CPUs can execute multiple threads simultaneously. A parallelised approach can also be distributed across multiple computers or cloud servers for further performance benefits; albeit being more complex due to the need to handle node failures, communication and distributed memory.

We plan to focus initially on a single task/language such as the Epsilon Validation Language (EVL) as proof-of-concept, and then abstract the execution semantics such that local-parallel and distributed-parallel solutions can be handled transparently in a more or less equivalent manner for other tasks. We will therefore initially target data-parallelism; where each thread executes the entire program over a subset of the model. This approach is arguably more scalable and is likely to provide better speedups because it minimizes the need for threads to communicate, as each thread can work independently without computational dependencies. This is especially relevant for distributed computing, where communication costs over a network may be much higher and/or undesirable.

We recognize that data dependencies are inevitable and that they pose a significant challenge for a data-parallel approach. We also understand that although parallelising the declarative parts of domain-specific languages is relatively straightforward, automatic parallelisation of imperative constructs is beyond the scope of this project. However, if time permits we may also consider parallel execution of lambda expressions (first-order logic operations) on collections by assuming them to be pure functions (i.e., without side-effects). Perhaps the most interesting challenges we expect to face will be with the implementation of model management tasks which mutate models; namely model transformations and model merging, since the level of granularity with regards to concurrent execution is likely to play a large role in the performance benefits and the overall complexity of the implementation.

With regards to GPU-accelerated execution, the greatest challenge is the notoriously limited and difficult programming model of graphics devices. Both CUDA and OpenCL frameworks require programs to be written in a subset of the C/C++ language; with limited support for non-primitive data types. Although the SIMD (Single Instruction Multiple Data) architecture of GPUs is ideal for our data-parallel approach, it is unclear whether the overhead of converting models and scripts to a compatible format for execution on GPUs, as well as transferring them to graphics memory, will outweigh the gains from increased parallelisation. A potential starting point for such an investigation would be in using MapReduce to define the execution semantics of a model management task (such as in [11]), since the use of MapReduce programming model is extensively studied in the GPU computing literature (for example, in [14], [15] and [16], just to name a few). Since MapReduce was originally designed for distributed computing applications, it may prove to be a more scalable solution overall and thus worth exploring; albeit the restrictive programming model may not be suitable for all model management tasks.

4 Preliminary Work and Expected Contributions

From a technical aspect, we have identified a number of ways to reduce the complexity of introducing concurrency in both local and distributed scenarios; namely JSCOOP [18] and Akka[3], as well as automatic parallelisation frameworks and techniques such as parallel streams, Hadoop[4] and Spark[5]. We have also investigated some automatic parallelisation for heterogeneous systems (i.e., applicable to both CPUs and GPUs), such as [19], [20] and [21].

So far, our implementation has focused on parallelisation of EVL based on previous work in an MSc project [22]; which identified many of the issues with concurrent execution of EVL programs and provided prototype solutions. Following a substantial amount of refactoring and further development, we have observed speedups up to 3x with four threads prior to any optimisations. Our current solution uses a fixed thread pool executor service, and the granularity of parallelisation is at the element level – that is, a new job is queued for each model element and constraint combination.

Whilst our research is broad in scope with many avenues for investigation, we endeavour to make significant contributions to the MDE community in the following areas:

- Thorough investigation of the challenges and approaches to executing model management programs concurrently
- A prototype parallel implementation of Epsilon supporting model validation, model comparison, model querying and model transformations
- A study of the similarities and differences between various model management tasks from a concurrent computation perspective
- An investigation of the extent to which GPU computing can be applied to accelerate the execution of model management programs
- An investigation of potential solutions for combining parallelism with incrementality and laziness for various model management tasks.

Our focus will be on model validation, model comparison and model-to-text transformations as these areas are deeply under-studied with regards to concurrent execution in the MDE literature.

5 Plan for Evaluation and Validation

To evaluate the scalability of our solutions, we intend on utilizing a common suite of models with varying sizes, ranging from hundreds to millions of elements. We will then compare speedups with the original single-threaded engines using multiple threads. We will also ensure that we test our solutions on hardware which can execute many threads simultaneously (e.g., 16, 32 or even more).

The domain-specific nature of modelling applications makes it difficult to obtain real-world models and programs from industry due to the intellectual properties. However, alternative sources such as [23] provide a reasonable starting point. More complex models can be obtained through the Train Benchmark

[3] `http://akka.io/` [4] `http://hadoop.apache.org/` [5] `http://spark.apache.org/`

[24], which allows for parametrised model generation. Although Epsilon supports the most commonly used model formats, it may prove challenging to find real–world *programs* to execute over these models, and even more difficult to find ones written in Epsilon languages. For models which have associated programs written in another language or suite (for example, transformations written in ATL or validations in OCL), we shall attempt to replicate the scripts as closely as possible in Epsilon. We shall also write scripts designed solely to test the functionality and performance of the execution engines to evaluate our implementation. Therefore, we aim to write scripts which not only use almost all features of the given language, but also write scripts of varying levels of complexity so that we can evaluate the effectiveness of a data-parallel approach as opposed to a rule-parallel one.

A further concern is that of correctness, since concurrent programs are notorious for non-deterministic behaviour. Although formal verification methods may theoretically be possible, they may be difficult to carry out for such large execution engines. At the very least, we plan to write a comprehensive suite of unit and system tests to build confidence of our implementation and results, ensuring that the outputs of the concurrent engines are identical to those of the original, non-concurrent engine implementations.

6 Current Status

This research is in its initial stages (less than six months in). We are currently working on laying down the foundations for making Epsilons engines thread-safe and identifying the parts which can be trivially parallelised. Our intention is to find the optimal balance between minimizing modifications and architectural changes to the existing engines whilst maximizing the performance gains. We expect the project to be complete by 2020; though we plan to publish any significant intermediate results for near-complete parallelised engines sooner.

References

1. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Scalability: The Holy Grail of Model Driven Engineering. In: MoDELS08 ChaMDE Workshop, pp.10-14 (2008).
2. Dávid, I., Ráth, I. & Varró, D.: Foundations for Streaming Model Transformations by Complex Event Processing. In: Software & Systems Modeling, pp. 1-28 (2016).
3. Tisi, M., Jouault, F.: Towards Incremental Execution of ATL Transformations. In: Proceedings of the Third international conference on Theory and practice of model transformations, pp. 123-137 (2010).
4. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proceedings of the 18th International Conference on Advanced Information Systems Engineering, pp. 81–95 (2006).
5. Ogunyomi, B.J. (2016): Incremental Model-to-Text Transformation, Doctoral thesis, University of York.
6. Razavi, A., Kontogiannis, K.: Partial Evaluation of Model Transformations. In: Proceedings of the 34th International Conference on Software Engineering, pp. 562-572 (2012).

7. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Lazy Execution of Model-to-Model Transformations. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, pp. 32–46 (2011).
8. Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems, Ottawa. pp. 46-61 (2015).
9. Burgueño, L., Troya, J., Vallecillo, A., Wimmer, M. (2015): Parallel In-place Model Transformations with LinTra. In: 3rd Workshop on Scalable Model Driven Engineering (BigMDE 2015).
10. Tisi, M., Martínez, S., Choura, H.: Parallel Execution of ATL Transformation Rules. In: Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems, pp. 656–672 (2013).
11. Benelallam, A., Gómez, A., Tisi, M., Cabot, J.: Distributed model-to-model transformation with ATL on MapReduce. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 37-48 (2015).
12. Dean, J., Ghemawat, S.: MapReduce: SimplifiedDataProcessingonLargeClusters. In: Sixth Symposium on Operating System Design and Implementation, pp. 137–149 (2004).
13. Benelallam, A., Tisi, M., Cuadrado, J.S., de Lara, J., Cabot, J.: Efficient model partitioning for distributed model transformations. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pp. 226–238 (2016).
14. Govindaraju, N.K., He, B., Luo, Q., Fang, W.: Mars: Accelerating MapReduce with Graphics Processors. In: IEEE Transactions on Parallel and Distributed Systems 22(4), pp. 608–620 (2011).
15. Hong, C., Chen, D., Chen, W., Zheng, W.: MapCG: writing parallel program portable between CPU and GPU. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques, pp. 217–226 (2010).
16. Xin, M., Li, H.: An Implementation of GPU Accelerated MapReduce: Using Hadoop with OpenCL for Data- and Compute-Intensive Jobs. In: Proceedings of the 2012 International Joint Conference on Service Sciences, pp. 6–11 (2012).
17. Martínez, S., Tisi, M., Douence, R.: Reactive model transformation with ATL. Science of Computer Programming 136(C), pp. 1–16 (2017).
18. Torshizi, F., Ostroff, J.S., Paige, R.F., Doyle, K.J., Lau, J. (2008): Jscoop: A high-level concurrency framework for java. Technical Report, York University.
19. Fumero, J.J., Steuwer, M., Dubach, C.: A Composable Array Function Interface for Heterogeneous Computing in Java. In: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, pp. 44–49 (2014).
20. Leung, A., Lhoták, O., Lashari, G.: Automatic parallelization for graphics processing units. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pp. 91–100 (2009).
21. Ishizaki, K., Hayashi, A., Koblents, G., Sarkar, V.: Compiling and Optimizing Java 8 Programs for GPU Execution. In: 24th International Conference on Parallel Architectures and Compilation Techniques, pp. 419–431 (2015).
22. Smith, M. (2015): Parallel Model Validation, MSc thesis, University of York.
23. `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra`
24. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The Train Benchmark: cross-technology performance evaluation of continuous model queries. In: Software & Systems Modeling, pp. 1-29 (2017).