Towards Declarative Imperative Data-parallel Systems [⋆]
– Discussion paper –

Matteo Interlandi[1] Giovanni Simonini[1] Sonia Bergamaschi[1]

Università di Modena e Reggio Emilia
firstname.lastname@unimore.it

Abstract. Pushed by recent evolvements in the field of declarative networking and data-parallel computation, we propose a first investigation over a *declarative imperative parallel programming model* which tries to combine the two worlds. We identify a set of requirements that the model should possess and introduce a conceptual sketch of the system implementing the foresaw model.

1 Introduction

Nowadays we are witnessing new trends such as *cloud computing* and *multicore processing* becoming popular. In fact hardware speed is not increasing anymore, therefore *distributed* and *parallel* architectures must be exploited in order to obtain performance improvements. However, it is well known that programming such kind of architectures is a very difficult task since they naturally give rise to non-deterministic (ND) outputs: the effect of a program over a mutable shared state may depend on the particular runtime interleaving of the execution flow. In fact, intuitively, the application in a ND order of a set of operations over a mutable shared state, may result in a ND outcome. The core of the problem can be summarized by the following equation:

$$\textit{ND Output} = \textit{ND Execution} + \textit{Mutable State} \tag{1}$$

Clearly, deterministic outputs can be obtained either (i) regualating the ND execution, or (ii) avoiding the mutable state. The first solution requires to explicitly reason over the *potential* parallel execution of operations – namely, their *concurrency*.

Imperative Languages and the Von Neumann Bottleneck: The main issue with *concurrent programming* is that to avoid ND output, an order over execution must be enforced. Knowing where and how this order must be injected is however an hard task, especially if an *imperative language* is used. To make this statement clear, we need to introduce the *Von Neumann Model* (VNM). In the VNM, computers are logically partitioned into memory, cpu, and a connecting tube. Then memory is again partitioned into a data segment and a code segment. This model suffers from an intrinsic limitation: the connection between the memory and the cpu acts as a bottleneck restricting the computation rate. This is what Backus named the *Von Neumann bottleneck* [8]. Due to such bottleneck, programmers are forced to split computation in sequences of operations that step by step must be applied by moving data and code back and forth

[⋆] Work taken from [16].

from the memory and the cpu. Imperative languages are high level abstractions of the VNM, and therefore they also inherit its bottleneck. In fact, for such languages, the program logic mainly concerns with the control flow, to assure that operations are executed in the proper order; while the program state is an ordered array of data values to permit an efficient access to memory locations. The main problem with imperative languages is hence that, due to their nature, they force programmers to conceptualize problems *sequentially in time*. Clearly, in this over-specified sequentiality it is difficult to understand where order is actually needed because required by the logic – and hence must be enforced by using proper constructs – from where, instead, parallel unordered executions can be exploited for better performances. To overcome the Von Neumann bottleneck, a different programming model must then be embraced: instead of specifying how the computation flow should proceed sequentially in *time*, programmers must be pushed to think more *in space*, i.e., computation intended as a set of order-agnostic transformations applied in parallel to a collection of input data elements. The output is then a new set of elements which can be used as new building blocks for successive computations [20]. This is exactly the idea pursued by the approach Backus suggested to surmount the Von Neumann bottleneck: *Functional Programming*.

Functional Languages and Immutable State: While ND outputs are avoided in imperative languages by explicitly imposing an order of execution through low level mechanisms, functional languages obtain the same result in a different way: state is *immutable* by default. As a consequence, the parallel nature of modern architectures can be fully exploited while maintaining a deterministic output. This is because the results of the operations, even if disorderly applied, can be accumulated and reconciled later using special purpose techniques [24] or application-specific logic. Thanks to the immutability of states, not only the Von Neumann bottleneck is avoided and *parallel programming* becomes natural, but also *fault-tolerance* concerns can be easily addressed: if only deterministic operations are considered, and the evolutions of the immutable states are logged, every time a state is lost because of a machine fault, it can be recomputed starting from the previous state, and replaying the proper set of operations [28]. Due to the above features, we are not surprised to see many data-parallel frameworks, such as *MapReduce* and *Spark*, embracing a functional programming style.

Data-Parallel Systems: If we put under analysis the plethora of large-scale data-parallel systems available nowadays, we can recognize two main approaches [21]: a *declarative* approach, pursued by parallel DBMS systems and based on the relational model; and an *imperative* approach, followed by modern "MapReduce-like" systems [13, 28] which are highly scalable, fault-tolerant, and mainly driven by industrial needs. Although there has been some work trying to bring together the two worlds, they mainly focus on exporting languages and interfaces – i.e., declarative languages on top of imperative systems [14], or MapReduce-like functions over parallel DBMS [27] – or in a systematic merging of the features of the two approaches [4]. Both approaches, however, share a common denominator: parallel specifications are compiled into a dataflow graph so that computation is not anymore driven (in time) by the progress of the program counter – as in the VNM – but (in space) by the availability of input data. We therefore advocate that a *declarative imperative* [15] approach for massive data-parallel system should be attempted: this is, data definition, manipulation and analytic can be expressed by means of a single declarative imperative language firmly grounded on the relational theory, with the execution framework following the same patterns commonly present in modern imperative data-parallel systems. Although many data-oriented systems are moving in this direction [5, 26], a comprehensive theoretical background is lacking [9]. Our goal is then to carry out a first step in this direction, by providing an investigation over the properties that the computational model, language and semantics of the envisioned data-parallel system should possess.

2 A Declarative Imperative Parallel Programming Model

Our starting point, as mentioned, is the relational model. More precisely, we focus on the branch of *logic programming* which intersects with the relation model: *Datalog*. Although the idea of employing a relational language to program a data-processing system appears quite natural – since it mainly concerns with data manipulation and movement – it is not clear how logic programming deals with the problem of distributed parallel computation summarized in eq. (1). It turns out that Datalog is in a very favorable position thanks to its "disordered" nature [5]. As we have previously explained, the problem with concurrent imperative programming is that maintaining order in modern architectures is complex. If we focus our attention on how a program state is represented, it is commonly acknowledged that the Von Neumann model is a particular implementation of a *Turing Machine* (TM), where the encoding of data on the tape is emulating the internal representation of programs states stored in the VNM memory. In both cases, such representation contains more information than the data itself, where extra information primarily deal with how the data is internally represented. In the relational model, instead, data is maintained into a database, where any extra information is discarded. The *data independence principle* exactly postulates this: a database provides an interface hiding the internal data representation [1]. This can be formalized at the computational level by means of *genericity*: computation is generic if insensible to automorphism of the data [2].

Datalog, though, is not only disordered for what concerns the data representation, but also on how transformations are applied to data. Programs, in fact, are composed by sets of rules which, operationally, can be applied to the input data in whichever sequence. This is because a unique deterministic outcome (also known as *minimal model*) will always be obtained, independently from a particular order of execution. Datalog is completely disordered and therefore a deterministic result is always obtained, whichever execution plan is non-deterministically selected.

Order and Mutable State: Nevertheless, Datalog is lacking of expressive power. Thus, if now we consider Datalog$^\neg$ programs (i.e., Datalog where negation is allowed), the uniqueness of the output is lost: different minimal models can be returned based on which order rules are applied to the input data. In such situation, in fact, more than one least fixpoint exists. Researchers solved such issue by selecting a *canonical* minimal model. We then have that different semantics have been developed based on which mechanism is employed to derive the canonical model. Interestingly, one of such mechanisms, i.e., *stratification*, consists in injecting an evaluation order over program rules.

By enforcing order, the unique minimal model semantics is restored. Interestingly, stratification is a syntactic concept: a stratification order can be inferred automatically by the compiler from a simple static analysis of the program. In some circumstances, however, programmers might be interested in directly reasoning over the ordered semantics of programs, e.g., for implementing iterative computation [11]. To achieve this, order can be promoted as a logical concept by augmenting Datalog$^{\neg}$ with a notion of *time*, interpreted as a monotonically increasing quantity. Embedding a notion of time directly into the language, mutable states become an emergent property of ordered execution [6], while the fault-tolerance of functional programming is maintained[1]. We can then start to look at eq. (1) from a more "logical" point of view: mutable state is not actually a factor determining the non-determinism of the output, but a consequence of the ordered semantics imposed on the program logic[2].

However, order is not only related with mutable states. The strict relationship between order and non-determinism is well known in the database community. Order, in fact, can be accomplished by breaking genericity and accessing the internal representation of data. From a conceptual point of view, this results in a ND behavior: given the same input different outcomes may appear on the basis of the accessed order[1]. On the other hand, non-determinism can be used to provide an arbitrary order while maintaining genericity, as accomplished, for example, by the `choice` [23] construct. Hence, we can now logically model the non-determinism of the parallel execution by suspending the data independence principle, and looking at the internal data representation. Of course, data will be organized in a ND order as a consequence of the parallel model in which data transformations are performed. The ND execution term of eq. (1) can then be expressed by employing the `choice` construct in a non-generic way, accessing the order in which data is derived [6]. Eq. (1) can now be rewritten, from a logical perspective, as:

$$\textit{ND Model} = \textit{ND Order} + \textit{Ordered Semantics} \tag{2}$$

The intuition behind equation (2) is that the complexity of programming parallel architectures actually resides in the interaction between the ordered semantics *intrinsic* in the program logic and the *incidental* ND order caused by the suspension of the data independence principle. In other words, if we switch to a spatial point of view, we have that the above complexity consists in the effort needed to correctly build the model required by the program logic, starting from the non-deterministic blocks generated de facto by the parallel execution. Now, intuitively, if only positive programs are considered, a deterministic output is always obtainable despite the non-deterministic way in which it is constructed. On the other hand, a similar deterministic behavior can be achieved by disallowing the access to the data representation, and enforcing generic computation. In this way, we can leave to the system the burden of understanding where the concurrent nature of the architecture can be fully exploited – e.g., for positive Datalog programs – and where, instead, more restrictive execution models must be imposed. This dichotomy between the logic and the control components [18] is exactly the same

[1] This because every state can be made instantaneously immutable – i.e., immutable inside one specific time-step – while over time intervals states appear as mutable.

[2] Or, equivalently, a consequence of the expressive power of the language.

approach pursued by database systems. For this reason, we name this solution *declarative parallel programming*.

Enforcing Determinism through Coordination: In recent years, a lot of emphasis has been placed on how distributed systems can exploit different levels of *consistency* in order to improve performances while maintaining correct deterministic outcomes [25]. Another topic highly related with consistency is *coordination*, usually informally interpreted as a mechanism which permits to achieve a distributed agreement on some property of a system. If consistency can be considered as such, coordination can be seen as a tool able to enforce consistency, when in the execution of a system this property is not in general achievable. We, therefore, consider coordination as the means used by the control component to enforce deterministic outputs. When coordination is required, the performance of parallel programs is affected, inasmuch as coordination guarantees deterministic outputs by imposing a sequential blocking execution. Therefore, being able to precisely assess which part of a specification can be evaluated in a non-blocking coordination-free way is of paramount importance. Modern dataflow systems usually inject blocking code at the physical level, when required by the semantics of the operators composing the plan [10]. In this way though, sequential execution is performed not because required by the program logic, but because needed by the particular implementation of the employed operators. On the contrary, we think that recent results linking monotonicity to coordination-freeness – i.e., the *CALM principle* [15] – must be exploited, so that blocking semantic is employed only when strictly required by the program logic.

Adding Imperative Analytics: Modern dataflow systems mix common database operations with complex analytical functions usually expressed in an imperative language [26]. How can our declarative programming model be integrated with an imperative one while maintaining all the developed theoretical properties? To reach this goal, a promising technique is to equip the language with user-defined functions (UDFs) [11]. Computation can then be conceptually modeled by means of an extension of *generic machines* (GMs) [2] we could name *generic transducers* (GTs). While GMs are TMs equipped with a relational store, we regard GTs as having the relational store substituted by a *relational transducer* [3]. In this way, we are able to scale the Abiteboul's model over distributed parallel architectures by defining a *network of GTs* as an enhancement of the *transducer network* model [7]. The rationale behind this approach is that, while the CALM analysis can be applied over the declarative part of the language, it is still not clear how such principle can be spread over imperative constructs. By employing such model, a proper set of interfaces acting among the transducer network and the Turing Machine can be developed, so that a CALM assessment can be performed thoroughly over the entire specification. The literature on user-defined aggregates could be of help in solving such concern: imperative functions extending streaming stateless interfaces à la Sawzall [22] compute only monotonic functions; streaming stateful interfaces with early returns can compute monotonic functions [29] – although the values distribution is not necessarily monotonic; while non-monotonic functions can be calculated only if extending a blocking stateful interface. To maintain the generic nature of computation, we require state updates to be *associative* and *commutative*, i.e., stateful interfaces must partially be ACID 2.0

3 The System

From the database literature, we know that relational database systems are commonly separated in *conceptual*, *logical*, and *physical* levels, where the data independence principle permits to hide low level physical details from the logical level, and the conceptual level is independent from a specific logical (and physical) representation of data. We are planning to apply a similar interpretation to our idea of declarative imperative systems. We deem such systems to be composed by four different layers: *logical*, *parallel*, *concurrent* and *physical*. At the logical layer, the main logic of the program can be expressed through a declarative imperative language. Here, not only complex machine learning functions can be implemented by properly extending one of the designed interfaces, but also different computational paradigms (e.g. MapReduce-like and graph-like) can be pipelined in a unique high-level program workflow. At this layer, no notion of parallel execution filters out from the underneath layers since *parallel specifications* are automatically synthesized from the program logic. At the parallel layer, the distributed parallel nature of the system is manifested as a *Synchronous Network* [17] of GTs (BSGTN): this is an abstraction providing a surrogate computational model emulating the behavior of common data-parallel systems, so that different computational models can be seamlessly adopted at the logical level. Data processing over a BSGTN proceeds in *super-steps*. Each super-step is composed by a *computation* and a *communication* phase, and a new super-step can only start when all the GTs have completed the current one, i.e., each GT is blocked until every other GT has reached the *synchronization barrier*. Parallel specifications are used to program the network of GTs. In analogy with the data independence and genericity principles, parallel specifications are required to be *independent* and *convergent*, in order to mask the parallel (and therefore also the logical) layer from a particular network instantiation and physical organization. The BSGTN computational model is implemented at the concurrent level. Here is where the monotonicity analysis is exploited to translate the synchronous (blocking) computation of the parallel layer into an asynchronous concurrent computation, where coordination code is automatically injected into a specification only when the blocking semantics is actually required by the program semantics. We are planning to use *Bloom* [5] to implement such layer. Thanks to Bloom, an optimized logical plan taking advantage of CALM can be derived from parallel specifications. CALM permits us to guarantee a smooth transition from the "always blocking" semantics of the synchronous computation of the parallel layer, to a "blocking only when needed" asynchronous computation at the concurrent level. In this layer, in addition, also non-functional concerns such as fault-tolerance and strugglers mitigation must be addressed. Finally, the physical level specifies the details about the dataflow physical plan and its distributed execution environment. This level implements the program-specific mechanisms for caching, storage and indexing. We consider REEF [12] as a proper tool for the physical layer.

4 Related Works

Our vision generalizes the approach of [11]. Differently from us, they depend on the physical layer, implemented using Hyracks, for the parallelization of programs. As a consequence, only sub-optimal physical plans can be created, since they are not able to fully exploit the properties – such as monotonicity – intrinsic in the declarative nature of specifications. We also share similarities with the *Bloom* language and its theoretical counterpart *Dedalus* [5, 6]. We focus on parallel programming and, thus we are mainly interested on how a deterministic output can be achieved by enforcing data independence in distributed parallel settings, whereas, Dedalus and Bloom target concurrent programming since they explicitly reason on non-deterministic executions. As already mentioned, we will exploit Bloom's features to implement our concurrent layer. Finally, note that in Bloom the interaction between the imperative and the declarative part has not yet been explored.

5 Conclusions

The power and simplicity of the relational database theory is impressive. We feel that parallel programming should take advantage of the relational model to set forth future declarative imperative data-parallel systems. We have envisioned such systems as being divided in four independent abstraction layers: logical, parallel, concurrent and physical. We propose a programming model based on Datalog (to be able to smoothly exploit CALM) augmented with imperative constructs (to model analytics functions). The interaction between the declarative and the imperative part will be performed through specifically tailored interfaces so that monotonic analysis can be performed thoroughly.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 209–219, New York, NY, USA, 1991. ACM.
3. S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 179–187, New York, NY, USA, 1998.
4. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, Aug. 2009.
5. P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.
6. P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors, *Datalog*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. 2010.
7. T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 283–292, New York, NY, USA, 2011.
8. J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.
9. K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, June 2009.

10. V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, 2011.
11. V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull.*, 35(2):24–32, 2012.
12. B.-G. Chun, T. Condie, C. Curino, C. Douglas, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, J. Rosen, R. Sears, and M. Weimer. Reef: Retainable evaluator execution framework. *Proc. VLDB Endow.*, 6(12):1370–1373, Aug. 2013.
13. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
14. A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, Aug. 2009.
15. J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.
16. M. Interlandi. *On Declarative Data-Parallel Computation: Models, Languages and Semantics*. PhD Dissertation. 2014.
17. M. Interlandi, L. Tanca, and S. Bergamaschi. Datalog in time and space, synchronously. In L. Bravo and M. Lenzerini, editors, *AMW*, volume 1087 of *CEUR Work. Proc.*. 2013.
18. R. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, July 1979.
19. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 97–108, New York, NY, USA, 2006. ACM.
20. M. Odersky. Working hard to keep it simple. http://www.oscon.com/oscon2011/public/schedule/detail/21055, July 2011.
21. A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, 165–178, 2009.
22. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, Oct. 2005.
23. D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 205–217, New York, NY, USA, 1990. ACM.
24. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA. 2011.
25. W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.
26. R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013.
27. Y. Xu, P. Kostamaa, and L. Gao. Integrating hadoop and parallel dbms. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 969–974, New York, NY, USA, 2010.
28. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for inmemory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012.
29. C. Zaniolo and H. Wang. Logic-based user-defined aggregates for the next generation of database systems. In K. Apt, V. Marek, M. Truszczynski, and D. Warren, editors, *The Logic Programming Paradigm*, Artificial Intelligence, pages 401–426. Springer, 1999.