A computational model for MapReduce job flow

Tommaso Di Noia, Marina Mongiello, Eugenio Di Sciascio

Dipartimento di Ingegneria Elettrica e Dell'informazione
Politecnico di Bari
Via E. Orabona, 4 – 70125 BARI, Italy
{firstname.lastname}@poliba.it

Abstract. Massive quantities of data are today processed using parallel computing frameworks that parallelize computations on large distributed clusters consisting of many machines. Such frameworks are adopted in big data analytic tasks as recommender systems, social network analysis, legal investigation that involve iterative computations over large datasets. One of the most used framework is MapReduce, scalable and suitable for data-intensive processing with a parallel computation model characterized by sequential and parallel processing interleaving. Its open-source implementation – Hadoop – is adopted by many cloud infrastructures as Google, Yahoo, Amazon, Facebook.
In this paper we propose a formal approach to model the MapReduce framework using model checking and temporal logics to verify properties of reliability and load balancingof the MapReduce job flow.

1 Introduction and motivation

During the last decades the phenomenon of "Big Data" has steadily increased with the growth of the amounts of data generated in academic, industrial and social applications. Massive quantities of data are processed on large distributed clusters consisting of commodity machines. New technologies and storage mechanisms are required to manage the complexity for storage, analyzing and processing high volumes of data. Some of these technologies provide the use of *computing as utility* – cloud computing – and define new models of parallel and distributed computations. Parallel computing frameworks enable and manage data allocation in data centers that are the physical layer of cloud computing implementation and provide the hardware the cloud runs on. One of the most known framework is MapReduce. Developed at Google Research [1] it has been adopted by many industrial players due to its properties of scalability and suitability for data-intensive processing. The main feature of MapReduce with respect to other existing parallel computational model is the sequential and parallel computation interleaving. MapReduce computations are performed with the support of data storage Google File System (GFS). MapReduce and GFS are at the basis of an open-source implementation Hadoop[1] adopted by many cloud infrastructures as Google, Yahoo, Amazon, Facebook.

[1] http://hadoop.apache.org

In this paper we propose a formal approach to model the MapReduce framework using model checking and tempora l logics to verify some relevant properties as reliability, load balancing, lack of deadlock of the MapReduce job flow. To the best of our knowledge only two works have combined MapReduce and model checking with a different aim from ours: in [2] MapReduce is adopted to compute distributed CTL algorithms and in [6] MapReduce is modeled using CSP formalism. The remaining of the paper is organized as follows. In Section 2 we recall basics of model checking and temporal logics, the formalism used to define and simulate our model. Section 3 provides a brief overview of the main features of MapReduce. Section 4 proposes our formal model of job flow in Mapreduce computation while Section 5 proposes an analytical model in the Uppaal model checker language with properties to be checked. Conclusion and future works are drawn in the last section.

2 Model Checking and Temporal Logics

The logical language we use for the model checking task is the Computation Tree Logic (CTL), a propositional, branching, temporal logic [5].

The syntax of the formulae can be defined, using Backus-Naur form, as follows (where p is an atomic proposition): $\phi, \psi ::= p \mid \phi \wedge \psi \mid \phi \vee \psi \mid \neg\phi \mid \phi \rightarrow \psi \mid \phi \leftrightarrow \psi \mid EF\phi \mid EX\phi \mid EG\phi \mid E(\phi U \psi) \mid AG\phi \mid AF\phi \mid AX\phi \mid A(\phi U \psi)$. An atomic proposition is the formula **true** or a ground atom CTL formulae can also contain path quantifiers followed by temporal operators. The path quantifier E specifies some path from the current state, while the path quantifier A specifies all paths from the current state. The temporal operators are X, the neXt-state operator; U, the Until operator; G, the Globally operator; and F the Future operator. The symbols X, U, G, F cannot occur without being preceded by the quantifiers E and A.

The semantics of the language is defined through a Kripke structure as the triple $(S, \rightarrow, L)$ where S is a collection of states, $\rightarrow$ is a binary relation on $S \times S$, stating that the system can move from state to state. Associated with each state s, the interpretation function L provides the set of atomic propositions $L(s)$ that are true at that particular state [3]. The semantics of boolean connectives is as usual. The semantics for temporal connectives is as follows: $X\phi$ specifies that ϕ holds in the next state along the path. $\phi U \psi$ specifies that ϕ holds on every state along the path until ψ is true. $G\phi$ specifies that ϕ holds on every state along the path. $F\phi$ specifies that there is at least one state along the path in which ϕ is true. The semantics of formulae is defined as follows: $EX\phi$: ϕ holds in some next state; $EF\phi$: a path exists such that ϕ holds in some Future state ; $EG\phi$: a path exists such that ϕ holds Globally along the path; $E(\phi U \psi)$: a path exists such that ϕ Untilψ holds on it; $AX\phi$: ϕ holds in every next state; $AF\phi$: for All paths there will be some Future state where ϕ holds; $AG\phi$: for All paths the property ϕ holds Globally; $A(\phi U \psi)$: All paths satisfy ϕ Until ψ. The *model checking problem* is the following: Given a model M, an initial state s and a CTL formula ϕ, check whether $M, s \models \phi$. $M \models \phi$ ehen the formula must be checked for every state of M.

3 MapReduce Overview and proposed model

MapReduce is a software framework for solving large-scale computing problems over large data-sets and data-intensive computing. It has grown to be the programming model for current distributed systems, i.e. cloud computing. It also forms the basis of the data-center software stack [4].

MapReduce framework was developed at Google Research as a parallel programming model with an associated implementation. The framework is highly scalable and location independent. It is used for the generation of data for Google's production web search service, for sorting, for data-intensive applications, for optimizing parallel jobs performance in data-intensive clusters. The most relevant feature of MapReduce processing is that computation runs on a large cluster of commodity machines [1]; while the main feature with respect to other existing parallel computational models is the sequential and parallel computation interleaving.

The MapReduce model is made up of the Map and Reduce functions, which are borrowed from functional languages such as Lisp [1]. Users' computations are written as Map and Reduce functions. The Map function processes a key/value pair to generate a set of intermediate key/value pairs. The Reduce function merges all intermediate values associated with the same intermediate key. Intermediate functions of Shuffle and Sorting are useful to split and sort the data chunks to be given in input to the Reduce function. We now define the computational model for MapReduce framework. We model jobs and tasks in MapReduce using the flow description as shown in Figure 1.

Definition 1 (MapReduce Graph (MRG)). *A MapReduce Graph (MRG) is a Direct Acyclic Graph* $G = \{\mathbf{N}, \mathbf{E}\}$*, where nodes* $\mathbf{N}$ *in the computation graph are the tasks of computation –* $\mathbf{N} = \mathbf{M} \cup \mathbf{S} \cup \mathbf{SR} \cup \mathbf{R}$ $(\mathbf{M} = map, \mathbf{S} = shuffle, \mathbf{SR} = sort, \mathbf{R} = reduce)$ *– and edges* $e \in \mathbf{E}$ *are such that:*

1. $\mathbf{E} \subseteq (\mathbf{M} \times \mathbf{S}) \cup (\mathbf{S} \times \mathbf{SR}) \cup (\mathbf{SR} \times \mathbf{R})$*, i.e. "edges connect map with shuffle, shuffle with sort and sort with reduce tasks";*
2. $e \in \mathbf{M} \times \mathbf{S}$ *breaks input into tokens;*
   $e \in \mathbf{S} \times \mathbf{SR}$ *sorts input tokens by type;*
   $e \in \mathbf{SR} \times \mathbf{R}$ *gives sort tokens to reducer.*
3. *Reduce sends input data for cluster allocation to the file system*

Definition 2 (MapReduce Task). *A MapReduce task* t *is a token of computation such that* $t \in (\mathbf{M} \cup \mathbf{S} \cup \mathbf{SR} \cup \mathbf{R})$.

Definition 3 (MapReduce Job). *A MapReduce Job is the sequence* $t^1 \to t^2 \ldots \to t^n$ *of MapReduce tasks where*

$$t_j^1 = (M_i, .., M_{i+p}) \to t_j^2 = S_k \to t_j^3 = SR_k \to t_j^4 = R_k$$

with $M_i \in \mathbf{M}$, $i = 1 \ldots n$, $S_j \in \mathbf{S}$, $SR_j \in \mathbf{SR}$, $R_j \in \mathbf{R}$, $j = 1 \ldots m$.

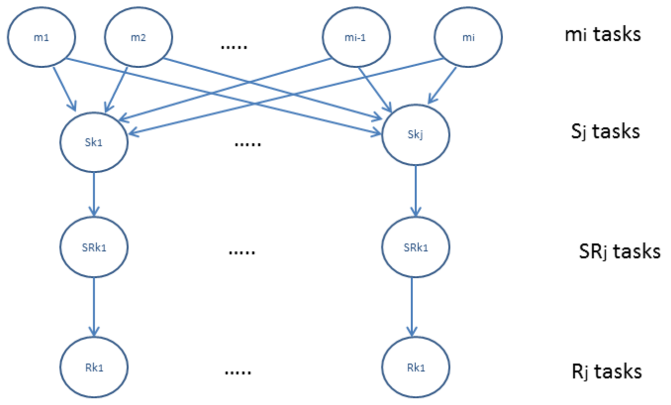

Fig. 1. MapReduce job flow model

4 Uppaal simulation model

In this Section we briefly describe the implemented model in the Uppaal[2] model checker's formalism. Uppaal model is described in XML data format of model checker description language and shown in the graphical interface of the tool as a graph model.

The analytical model is made up of three templates: job, mapper and Google File System (GFS). Figure 2 shows only the Uppaal graph model of the job described using the statechart notation. Main states of the job template are *Map*, *Shuffle*, *Sort* and *Reduce* as defined in the theoretical model in Section 3. Other states manage the flow of the job: *Starvation* models the state in which the task waits for unavailable resource, *Middle_state* manages exceptions, *receiving_input* and *receiving_keyval* manage data during job flow. Finally *medium_level* checks input data for the *Shuffle* state. Mapper template is made up of states: *Prevision* that checks the behavior of the mappers working for the given job. The *Prevision* state is followed by *Error* and *out_of_order* state in case of wrong behavior of the mapper, or *Work* state in case of correct behavior of the mapper. Finally the GFS template only manages the job execution.

To test the validity of the approach we simulated the model by instantiating a given number of jobs with relative mappers. We checked the two main properties of MapReduce framework, i.e. load balancing and fault tolerance.

Load Balancing. This property checks that the load of the Job J will be distributed to all tasks of the Mapper M_i with given map and reduce tasks. Hence when the job enters the state Map all the previson state of the mapper are veryfied, this means that the load is balanced between all the mappers.

$$EG(J.Map) \wedge AG(M_i.Prevision)$$

Fault Tolerance. If the M_i map is out of service the job mapper schedules an alternative task to perform the missed function that belongs to remaining

[2] http://www.uppaal.org/

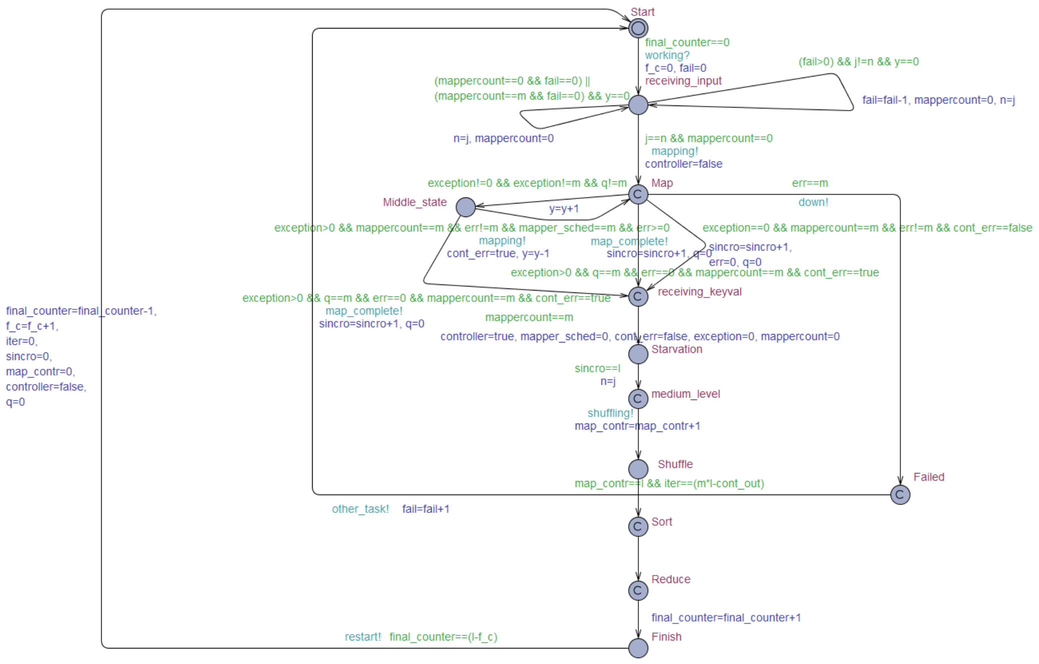
Start
final_counter==0
working?
f_c=0, fail=0
receiving_input
(fail>0) && j!=n && y==0
fail=fail-1, mappercount=0, n=j
(mappercount==0 && fail==0) ||
(mappercount==m && fail==0) && y==0
n=j, mappercount=0
j==n && mappercount==0
mapping!
controller=false
exception!=0 && exception!=m && q!=m
Map
err==m
down!
Middle_state
y=y+1
exception>0 && mappercount==m && err!=m && mapper_sched==m && err>=0
mapping!
cont_err=true, y=y-1
map_complete!
sincro=sincro+1, q=0
exception==0 && mappercount==m && err!=m && cont_err==false
sincro=sincro+1,
err=0, q=0
exception>0 && q==m && err==0 && mappercount==m && cont_err==true
exception>0 && q==m && err==0 && mappercount==m && cont_err==true
map_complete!
sincro=sincro+1, q=0
receiving_keyval
final_counter=final_counter-1,
f_c=f_c+1,
iter=0,
sincro=0,
map_contr=0,
controller=false,
q=0
mappercount==m
controller=true, mapper_sched=0, cont_err=false, exception=0, mappercount=0
Starvation
sincro==l
n=j
medium_level
shuffling!
map_contr=map_contr+1
Shuffle
map_contr==l && iter==(m*l-cont_out)
Failed
other_task!
fail=fail+1
Sort
Reduce
final_counter=final_counter+1
restart!
final_counter==(l-f_c)
Finish

Fig. 2. Uppaal simulation model

M_i. In the simulation model, $mapper_count$ is the counter of the total number of mappers and $mapper_sched$ is the variable that counts the scheduled mappers. The assigned value, m is the number of mappers.

$$EF(M.Out\_of\_order) \wedge AG(mapper\_count = m \wedge mapper\_sched = m)$$

From the simulation results we checked that the model ensures load balancing and fault tolerance.

5 Conclusion and future work

We proposed a formal model of MapReduce framework for data-intensive and computing-intensive environment. The model introduces the definition of MapReduce graph and MapReduce job and task. At this stage of the work we implemented and simulated the model with Uppaal model checker to verify basics properties of its computation as fault tolerance, load balancing and lack of deadlock. We are currently modeling other relevant features as scalability, data locality and extending the model with advanced job management activities such as job folding and job chaining.

We acknowledge support of project “A Knowledge based Holistic Integrated Research Approach” (KHIRA - PON 02_00563_3446857).

References

1. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters, OSDI04: Sixth symposium on operating system design and implementation, san francisco, ca, december, 2004. *S. Dill, R. Kumar, K. McCurley, S. Rajagopalan, D. Sivakumar, ad A. Tomkins, Self-similarity in the Web, Proc VLDB*, 2001.
2. Feng Guo, Guang Wei, Mengmeng Deng, and Wanlin Shi. Ctl model checking algorithm using mapreduce. In *Emerging Technologies for Information Systems, Computing, and Management*, pages 341–348. Springer, 2013.
3. M.R.A. Huth and M.D. Ryan. *Logic in Computer Science*. Cambridge University Press, 1999.
4. Krishna Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. *Computer Networks*, 53(17):2939–2965, 2009.
5. Edmund M.Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, Massachusetts, USA: MIT press., 1999.
6. Fan Yang, Wen Su, Huibiao Zhu, and Qin Li. Formalizing mapreduce with csp. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 358–367. IEEE, 2010.