Modeling replication and erasure coding in large scale distributed storage systems based on CEPH

Fluid approximation of pool depletion systems

Enrico Barbierato[1], Marco Gribaudo[1], and Daniele Manini[2]

[1] Dip. di Elettronica e Informazione, Politecnico di Milano,
via Ponzio 34/5, 20133 Milano, Italy,
[marco.gribaudo,enrico.barbierato]@polimi.it
[2] Dip. di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy,
manini@di.unito.it

Abstract. Today's most of high performance computing applications use parallel programming paradigms to reach the desired efficiency objectives. In particular, they divide the problem into small elements that can be solved in parallel by as many computing devices as available. Some examples are Apache Spark, the evolution of Hadoop and mapreduce, GPGPU (General Purpose Graphical Processing Units) applications, many-core and multi-core embedded systems. In many cases this type of applications can be modeled by pool depletion systems, that is queuing models characterized by a set of parallel servers whose goal is to execute a predetermined number of jobs. Although the modeling paradigm is very simple, it suffers from state space explosion, and can be used to model systems with a limited degree of parallelism only. The main contribution provided by this work consists of presenting a fluid approximation approach capturing the main features of the considered pool depletion systems and solving the above mentioned issues.

1 Introduction

High performance computing applications usually rely on parallel execution to divide a large problem in small elements that can be considered concurrently. In this way, the higher is the number of available computing resources, the faster the problem can be solved. In turn this allows to consider very large scale problems, such as searching images in a big database based on visual comparison, detecting anomalies in bank account usage or recommending users products based on the analysis of the choices made by a large set of costumers. Big Data and data-analytic applications rely on technologies that are based, for example, on Apache Spark [16], the evolution of Hadoop [1] and MapReduce [10] to parallelize computation and reduce communication overhead among the participating nodes. Spark has been proposed to supports applications reusing a working set of data across multiple parallel operations offering also the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs), i.e., a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Parallelization of tasks can also be exploited inside a single computer

or inside an embedded system. For example, technologies like GPGPU (General Purpose Graphical Processing Units) [14] allow the use of the computational feature of a graphic adapter to execute parallel tasks and have applications in video editing, image processing, cryptography and scientific calculus. Embedded systems are also based on many-core and multi-core CPUs, which sometimes are implemented in FPGA to push the use of parallelism at extreme levels.
In order to asses the scalability of the considered type of parallel application, modeling approach must be used since the acquisition of very large set of resources has high costs and could require long provisioning times. However, conventional modeling techniques cannot be applied due to size of the problem: standard techniques like Queuing Networks [12], Generalized Stochastic Petri Nets [13] or Performance Evaluation Process Algebras [11] suffer from state space explosion of the Continuous Time Markov Chains that they use to solve the problem. For this reason in the literature several different modeling approaches have been used to address this problem. For example, in [3] advanced simulation techniques have been used to consider large MapReduce applications, and the proposed approach has been extended in [4] to insert it in a multiformalism context in order address more complex applicative scenarios. Mean-field approach has instead been used in [8] to consider different types of BigData applications. In this work, the considered application is modeled by pool depletion systems where a given number of tasks must be completed by a set of computing resources. Indeed, they are queuing models characterized by a set of parallel servers whose goal is to execute a predetermined number of jobs. This paper presents i) the measurement performed on a real application to motivate the considered class of system, and ii) a more in-depth analysis on the effect of different task length distributions using a custom built discrete event simulator. In order to overcome the scalability problems caused by the state explosion arising when modeling such systems, a fluid model is proposed to approximate the average task ending time with a fluid variable representing the total number of tasks to be completed by the parallel servers. This model is finally exploited to study in an efficient way the execution times of Map-Reduce jobs.
The paper is structured as follows: in Section 2 a pool depletion systems is presented, and Section 3 shows how they can be can characterized in the considered scenario. The fluid model is described in Section 4, and it is exploited in Section 5 to analyze a MapReduce job. Section 6 concludes the paper.

2 Pool depletion systems

Pool depletion systems are models where a given number of tasks must be completed by a set of computing resources. Figure 1 shows a single class, single resource type pool depletion system. In this case, the system has to execute N tasks, each one requiring an independent, identically distributed service time according to a random variable $\mathbf{S}$. The system is composed by K identical servers: at the beginning, K tasks enters the system and start being served at the same time. The other $N - K$ are forced to wait in the external task pool. As soon as

the first task ends, another one is admitted from outside. The process is repeated until no more tasks are waiting in the pool; after that moment, some of the K server starts becoming idle. The job ends when all its task are finished. In this type of models, the study of the total depletion time (i.e. the time required to complete all N tasks and leave all the K servers of the system idle) can be very interesting. In [9], two classes, two resources, single server pool depletion systems where introduced to study energy consumption in large data-centers. In that paper, exponential service with processor sharing nodes were considered, and an analytical solution based on the generation of the state space was proposed.

In this work, which considers a single type of resource represented by K identical parallel servers, the complexity of the analytical solution depends on the chosen service time distribution $\mathbf{S}$. If the service time distribution is exponential, then the system has a relatively simple analytical solution. Let $\mu = 1/E[\mathbf{S}]$ denote the rate of the exponential distribution. For the first $N - K$ jobs, the inter-completion time corresponds to the time required by the first server to complete. Since all services are exponential, it corresponds to the minimum of K exponential distributions, which is again exponentially distributed with rate $K \cdot \mu$. When depletion starts, the inter-completion time is still exponentially distributed, but this time with rate $k \cdot \mu$ with $1 \leq k \leq K$. In the end, the total completion $\mathbf{C}$ time is distributed as the sum of K exponential distributions:

$$\mathbf{C} = Erlang(N - K, K \cdot \mu) + \sum_{k=1}^{K} Exp(k \cdot \mu). \tag{1}$$

An analytical expression for Equation 1 can be found for example in [2]. If $\mathbf{S}$ can be expressed as a Phase Type distribution (PH), an analytical solution is still possible, since the class of PH distribution is closed under both the minimum and the sum. However, the number of states can grows linearly with N, but exponentially with both K and the number of phases. In particular, following [7], where multiple servers with (PH) distribution and Markov Arrival Process where considered, let M denote the phases required to describe $\mathbf{S}$, then the total number of phases $\#PH$ required to express the distribution of $\mathbf{C}$ is:

$$\#PH = (N - K) \cdot \binom{K + M - 1}{M - 1} + \sum_{k=1}^{K} \binom{k + M - 1}{M - 1}. \tag{2}$$

Equation 1 shows that phase type distribution is not a feasible solution for realistic systems where the level of parallelism K can be in the $10^1 \tilde{1} 0^3$ range, and the number of phases required is also high due to the low coefficient of variation characterizing the task execution time distribution $\mathbf{S}$ in most applications.

In the following, the considered class of depletion models (both on measurements performed on a real parallel application run on a multi-core processor, and on discrete event simulation) is analyzed. Then a fluid model is proposed to capture the behavior of the average completion time in an efficient way.

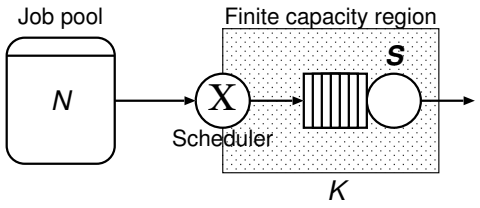

Fig. 1. A single class, single resource type pool depletion system.

3 Scenario characterization

The pool depletion model presented in this work can be applied to several parallel applications, ranging from Apache Spark [15], the evolution of Hadoop and map-reduce, GPGPU (General Purpose Graphical Processing Units) applications [14], many-core and multi-core embedded systems, and several parallel programming paradigms like the well known consumer-producer model. Firstly, this section presents measurement performed on a real application to motivate the considered class of system, secondly it illustrates an in-depth analysis on the effect of different task length distributions using a custom-built discrete event simulator.

3.1 Real system scenario

The performance model solution process used to perform what-if analysis on the models presented in [9] is considered as a benchmark for the class of systems described in this paper. The application considers the solution of a large Markov chain used to compute energy-related performance metrics for different parameters configurations. All the models are characterized by the same state-space, but by a different transition matrix. The algorithm generates several scenarios, then it solves them in a first-in-first-out queuing, running as many solution in parallel as available cores. Figure 2 shows the completion time of the considered benchmark on a quad-core, eight-threads MacBook Pro. In particular it considers the case with $K = 8$ simultaneous executions exploiting all the cores and all the hardware threads of the CPU, and the case with $K = 4$ avoiding the use of multithreading limiting the parallelism to the number of available cores. On the y-axis, the number of remaining tasks is presented, while the x-axis shows the ordered times at which tasks complete. The intersection of the curves with the x-axis defines the total job execution time. Even if all runs use the same parameterization, they are characterized by a different running time: this depends on the operating system and on the energy saving configuration of the machine slowing down in traces in non-deterministic ways. However, all cases have a ladder-like shape, with the height of a stair corresponding to the K, the level of parallelism (this is a direct consequence of the fact that all tasks are related to the same model, and are characterized by a very similar running

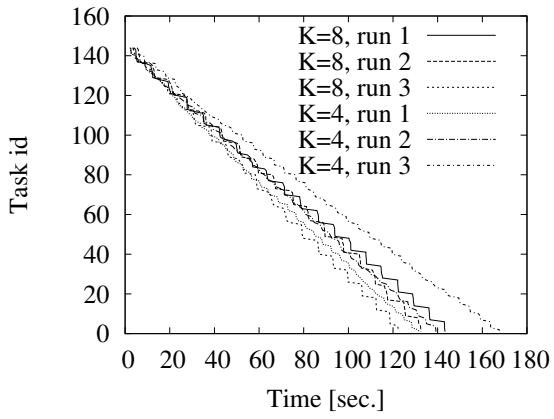

Fig. 2. Execution times of the considered benchmark on a quad-core, eight-threads MacBook Pro for different runs with and without multi-threading.

time). The runs with $K = 4$ requires more or less the same time and present the same ladder pattern, even if in this case the steps are smaller and shorter. This is due to the fact that for the considered benchmark the CPU multithreading is not able to provide significant performance increases, determining very similar performances to the $K = 8$ cases.

Figure 3 shows the average execution time μ with $N = 18$ tasks running on $K = 8$ simultaneous threads over 100 experiments. The standard deviation σ of the execution time is shown added to (curve $\mu + \sigma$) and removed from (curve $\mu - \sigma$) the average. The coefficient of variation (cv) is shown on the secondary axis. As it can be seen, despite the variability outlined in Figure 2, the cv is very small, and tends to decrease as the number of tasks increases, with jump only corresponding to the stairs of the ladder-shaped evolution of the average.

The previous results are confirmed in Figure 4 showing the distribution of the execution time of the benchmark application running $N = 18$ tasks on $K = 8$ simultaneous threads for the first three, the ninth and the last task. The small shift in the distribution of the ending time of the first three tasks emphasizes the little difference in execution time in the considered blocks, while the last tasks show that there is an expected spread in the completion time, even if the increase in the standard deviation is smaller than the one of the mean.

Note that this behaviour is characteristic of most the applications to which pool depletion models applies: for example, map-reduce techniques have the goal to produce small chunks of similar execution times. GPGPU applications instead base their parallelism on SIMD (single-instruction-multiple-data) techniques, which do not allow the different tasks to have significant differences in their execution times.

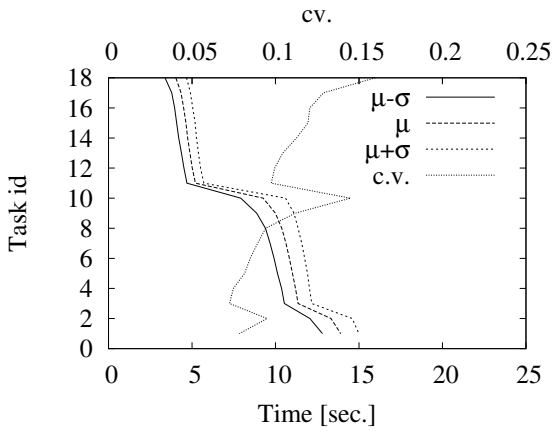

Fig. 3. Average execution time, standard deviation and coefficient of variation (cv) of the benchmark application running $N = 18$ tasks on $K = 8$ simultaneous threads.

3.2 Impact of task-length distributions

An analytical scenario is now considered to study the impact of different task duration distributions, all characterized by the same average. The focus concerns different distributions characterized by the same variation coefficient (where possible). For $cv = 1$ an exponential and a log-normal distribution is taken in account. For $cv = 0.5$ and $cv = 1/6$ two Erlang (respectively with four and thirty-six stages), two uniform and two log-normal distributions are considered, including $cv = 2$ with a Hyper-exponential and a log-normal distribution to have an idea of the impact of $cv > 1$. Larger cv are ignored since they are outside the intended range of applications. As a limiting case, the effect of the deterministic distribution ($cv = 0$) is shown as well. All curves have been computed with discrete event simulation. Although confidence intervals have been computed, they have not been shown for the sake of simplicity. Figure 5 shows the average execution time for the considered distributions of the task duration. In particular, Figure 5a-c) focuses on the case with $K = 20$ and $n = 144$. The whole picture of the job ending times is given in Figure 5a. The distribution with $cv \leq 1$ tends to have a very similar behaviour for the average time at which the single tasks complete. Distributions with $cv > 1$ tends instead to finish the first task earlier, while having a longer job completion time. The most important result that can be appreciated is that the average ending time of each task is mainly influenced by the cv: higher moments of the distribution play an impact only for the very first completion times, and tend to become less and less evident as time passes. Figure 4b zooms on the ending times of the first tasks. As it can be seen, only the deterministic component has a clear ladder behaviour. As the cv increases and as jobs complete, it become less and less evident. Moreover, after the very few tasks, the effect of higher moments also disappears, and curves of distributions with the same cv overlaps almost perfectly. The last tasks ending times

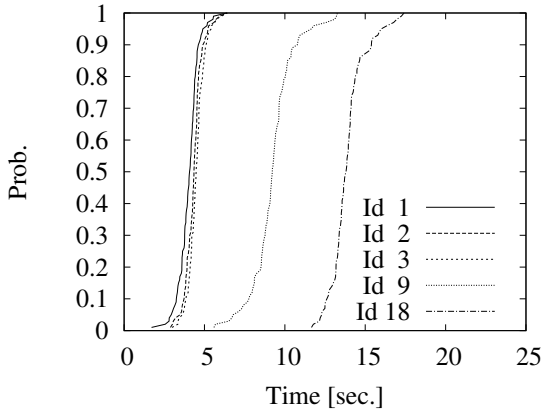

Fig. 4. Distribution of the execution time of the benchmark application running $N = 18$ tasks on $K = 8$ simultaneous threads for the first three, the ninth and the last task.

are instead considered in Figure 5c. As it can be seen, as depletion occurs, the higher moments play a significant role again, by creating slightly different tails. To better emphasize the effect of distribution with a $cv > 1$, Figure 5d considers the case with $K = 80$ and $N = 3600$. When a large number of tasks has to be executed, the task length distribution plays a marginal role, and all scenario evolves at a rate which can be estimated to $E[\mathbf{S}]/K$. The cv performs a shift in the various curves, and determines the speed at which the depletion moves away from the considered average behaviour. This type of evolution motivates the idea of resorting to fluid models to efficiently consider this type of systems.

The effect of the level of parallelism K is studied in Figure 6 showing the average execution time for the exponential and for a 36 stages Erlang distribution (corresponding to a $cv = 1/6$) required to run 720 tasks for $K \in \{8, 20, 40, 80\}$. The length of the tail of the distribution becomes more important as K increases, and also the effect of the cv becomes more evident. The staircase curve is destroyed very earlier in the execution, and becomes negligible after $4 \cdot K$ tasks have been completed even for a relatively low cv.

Figure 7 shows the distribution of the execution time for several tasks duration distributions. The exponential distribution, the four-stage Erlang distribution ($cv = 1/2$) and the Hyper-Exponential distribution (with $cv = 2$) and three different instances of a log-normal distribution with the same cv as the other are shown. All the curves refer to the completion job time when it is split into $N = 144$ tasks and it is run with parallelism level $K = 8$. Except for the case with $cv = 2$, where the two distributions have very different shapes, the distribution with the same cv presents very similar behaviours, which tend to differ for what concerns the probability of having larger completion times.

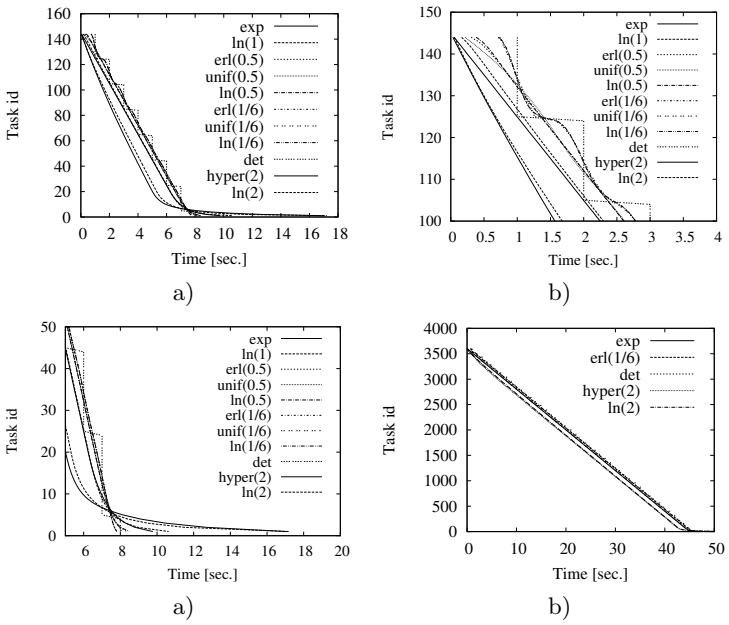

Fig. 5. Average execution time for different task duration distributions: a), b) and c) shows the respectively the complete, the detail of the first tasks and the detail of the last tasks for the case with $K = 20$ and $N = 144$; d) considers the case with $K = 80$ and $N = 3600$.

4 Fluid model

Due to the characteristic evolution of the completion time of the considered task, we propose to approximate the average task ending time with a fluid variable. In particular, a fluid variable x starting with the total number of tasks N that needs to be completed by the K parallel servers is added. The continuous variable then decreases at a fluid dependent rate that mimics the behaviour presented in Section 3. Following [5], a fluid model is considered where the evolution of the continuous variable is defined by a function $\phi : \mathbb{R}^2 \rightarrow$, where $\phi(x,t) = x'$ represents a system which has an average of x jobs to be completed at time 0, will remain with an average of x' jobs to be completed at time t. The fluid evolution function must satisfy the property that if the value of the continuous variable is $x(t_a)$ at a time instant t_a, then it must be that $x(t_b) = \phi(x(t_a), t_b - t_a)$. This is achieved in the following way: let $\mu(t)$ denote the average number of tasks in the system at time t. For example, $\mu(t)$ could correspond to one of the curves shown in Figure 5. Then, it is possible to say that:

$$\phi(x,t) = \mu(t + \mu^{-1}(x)) \tag{3}$$

If the fluid variable starts from a point $x(0) \neq \mu(0)$ at time $t = 0$, the definition of $\phi(x,t)$ shifts the fluid evolution curve to allow the continuity of $x(t)$. The

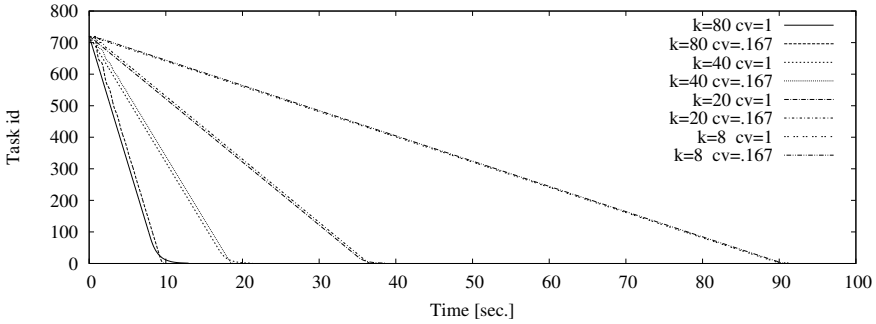

Fig. 6. Distribution of the execution time for the exponential and the 36 stages Erlang distribution $cv = 1/6 \approx 0.167$, for a different level of parallelism $K \in \{8, 20, 40, 80\}$ and $N = 720$ tasks.

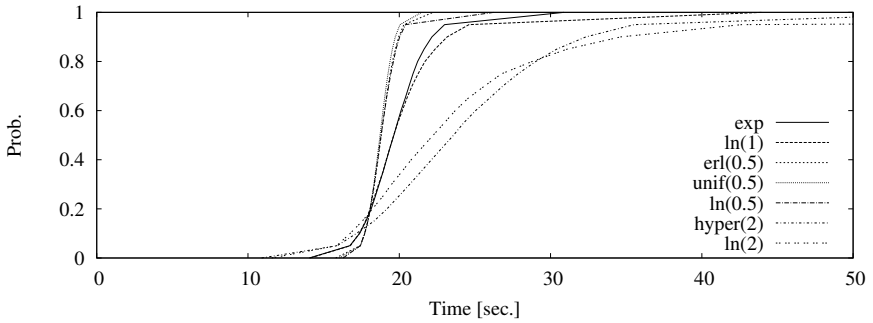

Fig. 7. Distribution of the execution time for several task duration distributions on a system running $N = 144$ tasks with a parallelism level $K = 8$.

average task completion time function $\mu(t)$ can be computed in many ways: for example, it can be determined from measurements taken on a real system. This is shown for example in Figure 3. In the following, $\mu(t)$ is computed with discrete event simulation, and then by performing linear interpolation among the task completion times.

5 Case study: analysis of Map-Reduce completion times

This section exploits the fluid model outlined in Section 4 to study in an efficient way the execution times of Map-Reduce jobs. *Map-Reduce* is a paradigm that allows to create applications able to compute huge amount of data in parallel, and Apache Hadoop is an example of a framework allowing to exploit it. The basis of the framework is a functional programming where data are not shared among threads but are passed as parameters or return values. An application running Map-Reduce has to specify input and output files, and map and reduce functions.

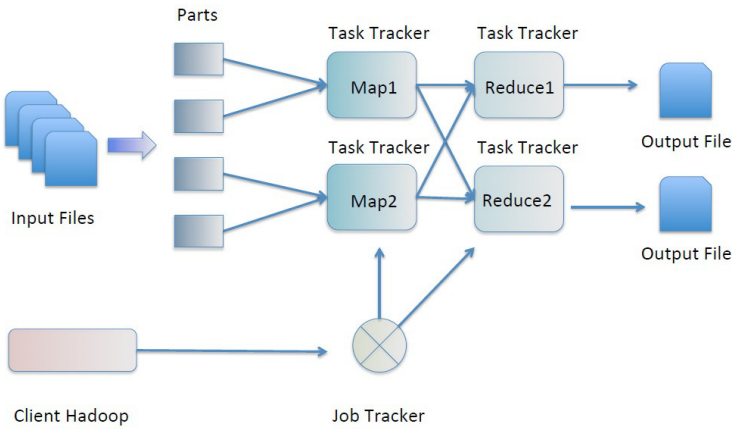

Fig. 8. Logic Scheme of Map-Reduce.

The Hadoop client exploits the service provided by the JobTracker that farms out MapReduce tasks to specific nodes in the cluster, ideally the nodes having the data, or at least are in the same rack. A TaskTracker is a node in the cluster accepting tasks (i.e. Map, Reduce and Shuffle operations) from a JobTracker. In particular, the Hadoop client provides the jobs and the configurations to the Job Tracker distributing them to the nodes for the execution. The Job Tracker also defines the number of parts by which the input files must be split into, and activates the Task Trackers according to their distance to the nodes holding the respective data. Task Trackers get the data to manage and activate the map function, then they run the reduce phase where data are sorted and aggregated. Then the output is generated and saved in a different file for each tracker. Figure 8 shows a logical scheme of execution of Map-Reduce jobs.

One of the key-features of Map-Reduce is that resources can be acquired dynamically during the execution of the tasks: as more computation nodes become available, they start working on the parts until all have been considered. After all the parts of the map phase have finished, the reduce stage can start. From a modeling point of view, a Map-Reduce job can be considered as the sequence of two pool depletion models representing respectively the map and reduce stages, where the number of available resources K changes with time. Although starting the next job while the previous one is completing can increase the throughput of the system, it complicates a lot the models that should be used to correctly capture the evolution of the system. Thanks to the proposed fluid technique, accurate estimates can be obtained with very simple models.

5.1 A Fluid Petri Net model of Map-Reduce

Figure 9 shows a Fluid Petri Net model describing the considered Map-Reduce paradigm, where resources are obtained gradually. The place named `Start`, initially marked, represent the start of the job. At time $t = 0$, immediate transition `Mstarts` fires, moving the token to place `Map` to denote the execution of the map phase. Fluid place `Tasks` represents the count of tasks that needs to be executed. When the map phase starts with the firing of `Mstart`, a set arc inserts into place `Tasks` the number of tasks that will be executed in that stage. Fluid transition `Depletion` models the execution of tasks. It behaves according to the definitions given in Section 4 in a state dependent way as it will be described later. The fluid arc that connects place `Tasks` to transition `Depletion` removes tasks as they finish in a continuous way. As soon as the tasks of the map phase end, transition `Rstart` fires, moving the token into place `Reduce` to denote the beginning of the reduce stage. Again, the set arc connecting transition `Rstart` to place `Tasks` defines the tasks that have to be executed in the reduce phase. When also the reduce phase ends, the model stops with the firing of immediate transitions `End`. The number of nodes running tasks in parallel is modeled by place `Nodes`. Acquisition of resources is modeled by transition `Nadd`, which stop firing thanks to the inhibitor arc coming from place `Nodes` as soon as the maximum number of available nodes K is reached. In the reduce phase, resources can be released due to the firing transition `Nrelease`. This transition is guarded by a function that depends both on the number of remaining tasks, and on the number of available resources. When the marking of `Tasks` becomes smaller that the marking of `Nodes`, a node can be released by allowing transition `Nrelease` to fire. Resources are also released when the reduce phase ends with the flush-out arc connecting place `Nodes` to transition `End`. The state-dependency of transition `Depletion` is characterized by $2K$ fluid evolution functions $\mu_{mr,K}(x,t)$, one for each combination of number of available nodes K, and map or reduce stage mr. The marking of places `Nodes`, `Map` and `Reduce` determines which fluid evolution function should be used in that particular moment of the model evolution.

5.2 Results

Figure 10 shows the results obtained with the solution of the FSPN presented in Figure 9. In this case we consider that $K = 20$ nodes have to process $N_{Map} = 144$ map tasks, and $N_{Reduce} = 192$ reduce tasks. The duration of map tasks $\mathcal{S}_{Map} \sim Erlang_4(1000)$ is assumed to be Erlang distributed with an average of 1000ms, and a $cv = 0.5$. Reduce tasks $\mathcal{S}_{Reduce} \sim Erlang_4(500)$ are instead assumed to be still Erlang distributed with $cv = 0.5$, but this time with an average of 500ms. Nodes are acquired at regular deterministic time intervals, with a new resource being available every 400ms. As introduced in Section5.1, the model exploits $2K$ fluid evolution functions $\mu_{mr,K}(x,t)$. In this example, such functions are determined by a quick run of discrete event simulation: due to the small cv that characterizes the considered Erlang distributions, acceptable confidence intervals can be achieved in about 100 runs per number of core and map or reduce stage.

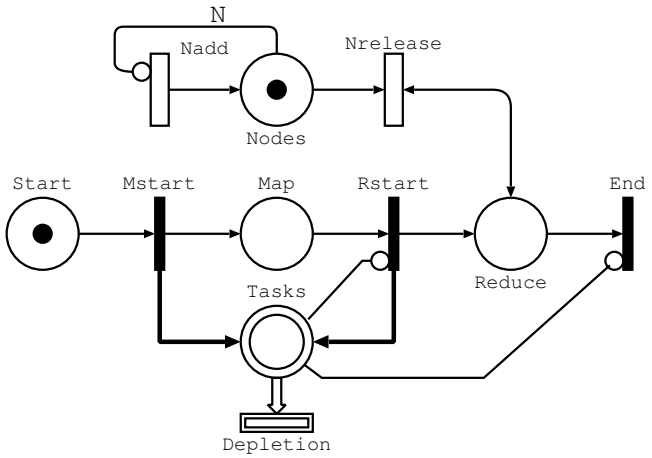

Fig. 9. A Fluid Petri Net model of the Map-Reduce paradigm.

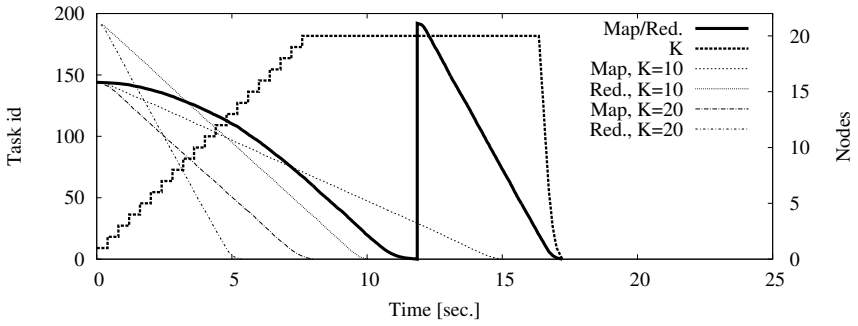

Fig. 10. Results of the FSPN model.

Figure 10 shows the fluid evolution function for both map and reduce tasks with $K = 20$ (the maximum number of available resources), and $K = 10$. The bold dotted line, referred to the secondary axis, shows the number of available cores as the function of time, while the curve with the largest width represents the fluid evolution of place `Tasks`. As it can be seen, as soon as number of cores reaches the maximum available $K = 20$, the evolution corresponds to the corresponding fluid models, translated to make the time evolution continuous. When the reduce phase reaches the end, cores starts be released, ad the curve corresponding to K drops very quickly to zero. In the considered scenario, the average time estimated to complete a map-reduce job is $R = 17.219$ sec.

5.3 Validation

To remark the validity of the proposed technique, the obtained results are compared against a simulation of the system performed in JMT [6] using the model shown in Figure 11. The system is modeled by a closed queuing network with a single job circulating in it. A delay station `TinyTerminalDelay` with a negligible waiting time is used as a reference station for the single circulating job. As soon as the single job leaves the reference station, it is split in two by the `JobFork` station: the upper job represents the execution of the map-reduce, while the bottom one models the acquisition of the resources. Again, a negligible delay `TinyMapDelay` is applied to the upper path, to allow the system to setup the available resources before starting to work on the tasks. The N_{Map} tasks are generated by the `MapFork` primitive. The execution of the tasks is modeled by the delay station `Map`, characterized by service time distribution $\mathcal{S}_{Map}$. As soon as all the map tasks are completed, they are united into a single entity by the `MapJoin` station, and then they are immediately split again into N_{Reduce}. Reduce tasks are executed by the delay station `Reduce`, whose service time distribution corresponds to $\mathcal{S}_{Reduce}$. The gradual acquisition of resources is modeled by `ResourcesFork` that inserts in the system $K-1$ jobs that are immediately routed to the FIFO queue station `ResoruceUsed`. From this station, jobs completes every 400ms. Resources constraints are modeled by the finite capacity region `FCR` that includes stations `Map`, `Reduce` and `ResourceUsed`, and that is characterized by total capacity K. In this way, at the beginning, only one extra task is allowed to enter the FCS from the `MapFork` station. As customers leaves the `ResourceUsed` stations, new computation nodes become available, and new tasks are allowed to start being served. Stations `ReduceJoin`, `ResourceJoin` and `JobJoin` are used to restore the single job and return it to the reference station. In this model, the average response time corresponds to the average time to complete a map-reduce job, with the considered resource acquisition policy, and should correspond to the one computed by the FSPN shown in Figure 9. Running JMT, we have obtained an average response time $R = 17.528$ sec, which differs from the result obtained with the FSPN model for 1.78%. This shows that the fluid approach can indeed provide good approximation, at a much lower computational effort.

6 Conclusions

This paper has proposed a characterization of single node, single class, multiple server, pool depletion system, and has shown how the average completion time of this type of models can be efficiently described by a fluid approximation. The importance of this type of system has been proven by applying the proposed technique to study incremental resource acquisition in map-reduce task execution. Future work will exploit the proposed characterization to define the fluid model starting from system parameters like the number of nodes, the number of tasks, and the average and coefficient of variation of the service time distribution. Moreover, the proposed technique will be used to study the completion

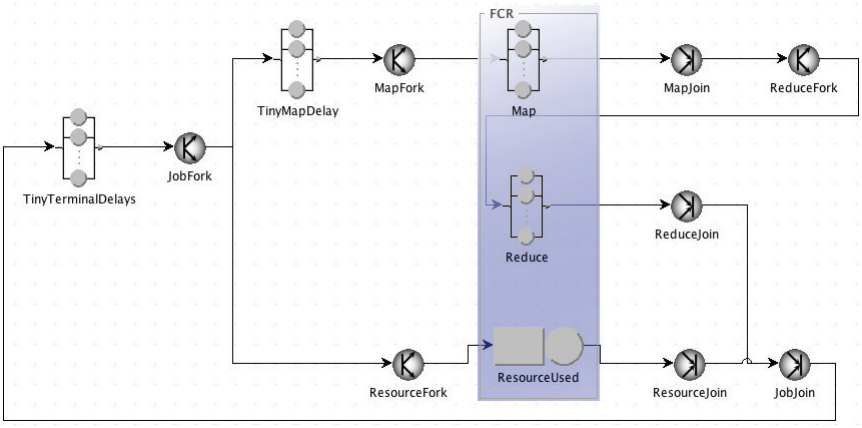

Fig. 11. A Fork/Join with Finite Capacity Regions Queuing Network model of the Map-Reduce paradigm, used to validate the results obtained with the FSPN model.

time in Spark jobs characterized by more complex fork-join structures and task execution policies. Thanks to their efficiency, fluid model will be exploited into optimization algorithm to dynamical configure the system by setting the proper number of nodes, number of tasks and tasks duration to reach proposed KPI with the lowest possible expense in terms of utilization and energy consumption.

References

1. Hadoop website. `https://hadoop.apache.org/docs/r1.0.4/` (2013)
2. Amari, S.V., Misra, R.B.: Closed-form expressions for distribution of sum of exponential random variables. IEEE Transactions on Reliability 46(4), 519–522 (Dec 1997)
3. Barbierato, E., Gribaudo, M., Iacono, M.: A performance modeling language for big data architectures. In: Proceedings of the 27th European Conference on Modelling and Simulation, ECMS 2013, Ålesund, Norway, May 27-30, 2013. pp. 511–517 (2013), `http://dx.doi.org/10.7148/2013-0511`
4. Barbierato, E., Gribaudo, M., Iacono, M.: Performance evaluation of nosql big-data applications using multi-formalism models. Future Generation Computer Systems 37, 345 – 353 (2014), `http://www.sciencedirect.com/science/article/pii/S0167739X14000028`, special Section: Innovative Methods and Algorithms for Advanced Data-Intensive ComputingSpecial Section: Semantics, Intelligent processing and services for big dataSpecial Section: Advances in Data-Intensive Modelling and SimulationSpecial Section: Hybrid Intelligence for Growing Internet and its Applications
5. Barbierato, E., Gribaudo, M., Iacono, M.: Modeling hybrid systems in SIMTHESys. In: Proceedings of the 8th International Workshop on Practical Applications of Stochastic Modelling. to appear (2016)
6. Bertoli, M., Casale, G., Serazzi, G.: Jmt: performance engineering tools for system modeling. SIGMETRICS Perform. Eval. Rev. 36(4), 10–15 (2009)

7. Bodrog, L., Gribaudo, M., Horvth, G., Mszros, A., Telek, M.: Control of queues with map servers: experimental results. In: Matrix-Analytic Methods in Stochastic Models: MAM8 - 2016, Kerala, India, January 8-10, 2014, Proceedings. pp. 9–11 (2014)
8. Castiglione, A., Gribaudo, M., Iacono, M., Palmieri, F.: Exploiting mean field analysis to model performances of big data architectures. Future Generation Comp. Syst. 37, 203–211 (2014), http://dx.doi.org/10.1016/j.future.2013.07.016
9. Cerotti, D., Gribaudo, M., Pinciroli, R., Serazzi, G.: Stochastic analysis of energy consumption in pool depletion systems. In: Measurement, Modelling and Evaluation of Dependable Computer and Communication Systems - 18th International GI/ITG Conference, MMB & DFT 2016, Münster, Germany, April 4-6, 2016, Proceedings. pp. 25–39 (2016), http://dx.doi.org/10.1007/978-3-319-31559-1_4
10. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Communications of the ACM - 50th anniversary issue: 1958 - 2008 51(1), 107–113 (2008)
11. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, New York, NY, USA (1996)
12. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1984)
13. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley & Sons, Inc., New York, NY, USA, 1st edn. (1994)
14. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. Queue 6(2), 40–53 (Mar 2008), http://doi.acm.org/10.1145/1365490.1365500
15. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. pp. 10–10. HotCloud'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1863103.1863113
16. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP 2013. pp. 423–438. ACM (2013)