Modeling the Performance of the Hadoop Online Prototype

Emanuel Vianna, Giovanni Comarela, Tatiana Pontes, Jussara Almeida, Virgílio Almeida
Department of Computer Science – Federal University of Minas Gerais (UFMG) – Brazil
{vianna, giovannicomarela, tpontes, jussara, virgilio}@dcc.ufmg.br

Kevin Wilkinson, Harumi Kuno, Umeshwar Dayal
Hewlett Packard Laboratories (HP-Labs) – Palo Alto, CA - USA
{kevin.wilkinson, harumi.kuno, umeshwar.dayal}@hp.com

Abstract

MapReduce is an important paradigm to support modern data-intensive applications. In this paper we address the challenge of modeling performance of one implementation of MapReduce called Hadoop Online Prototype (HOP), with a specific target on the intra-job pipeline parallelism. We use a hierarchical model that combines a precedence model and a queuing network model to capture the intra-job synchronization constraints. We first show how to build a precedence graph that represents the dependencies among multiple tasks of the same job. We then apply it jointly with an approximate Mean Value Analysis (aMVA) solution to predict mean job response time and resource utilization. We validate our solution against a queuing network simulator in various scenarios, finding that our performance model presents a close agreement, with maximum relative difference under 15%.

1 Introduction

MapReduce [2] is a popular and simple programming model for performing parallel computations on large data sets. A number of applications in data mining, machine learning and graph processing are suitable to its simple interface. Several implementations of the MapReduce model were developed, including Hadoop [4], and are being widely adopted in research and enterprises [3].

Recently, an extension of Hadoop called Hadoop Online Prototype (HOP) [1] was proposed. HOP provides pipeline parallelism among its tasks, which was not supported originally by Hadoop. The desire to understand the performance of MapReduce systems has led to a variety of efforts [5, 1, 14]. However, to the best of our knowledge, there is no previous work in modeling the performance of the pipeline parallelism inherent to HOP jobs.

HOP workloads are composed of a set of jobs, each one containing a set of tasks. Besides the execution time, these tasks may experience two types of delay: (1) queuing delays due to contention at shared resources, and (2) synchronization delays due to precedence constraints among tasks [8]. The literature is rich of modeling techniques to predict performance of workloads that do not exhibit synchronization delays. In particular, Mean Value Analysis (MVA) [10] has been applied to predict the average performance of several applications in various scenarios [7, 13]. However, MVA cannot be directly applied to workloads that have precedence constraints, such as HOP applications.

In this context, the goal of this paper is to develop a reasonably accurate analytical model to predict performance metrics (e.g. average job response time) of HOP workloads. The understanding of system performance under different circumstances is key to critical decision-making in workload management and capacity planning, such as: how many and what type of machines are required in a commodity cluster to support the Service Level Agreement (SLA)? How should HOP parameters be fine-tuned?

We here present an analytical performance model that captures the intra-job pipeline parallelism inherent to a HOP job. Our solution is built from a previous model [8], referred to as reference model, which was designed to predict the performance of parallel computations. Given a tree specifying the precedent constraints among tasks of a parallel job, this reference model applies an iterative approximate Mean Value Analysis (aMVA) algorithm to predict performance. The reference model allows different types of precedence constraints among tasks of a job, specified by simple task operators, as will be discussed in Section 2.1. However, none of these operators capture the precedence constraints of a pipeline. Thus, the reference model cannot be directly applied to HOP workloads.

Our solution, which extends the reference model, represents the intra-job pipeline inherent to HOP jobs as a prece-

1550-6533/11 $26.00 © 2011 IEEE
DOI 10.1109/SBAC-PAD.2011.24

dence tree, built using the simple task operators introduced in [8]. Our contributions over [8] are twofold: we explicitly address pipeline parallelism, and show how to use the primitive task operators introduced in the reference model to build a precedence tree for it, and (2) we also propose to use a different strategy to estimate the response time of subsets of a HOP job's tasks which leads to more accurate estimates of job average response time. We validate our model against an event-driven queuing network simulator, finding a very close agreement, with maximum relative difference under 15%.

The remaining of this paper is organized as follows. Related work is presented in Section 2. The HOP architecture and workloads are described in Section 3. Section 4 presents our analytical model, whereas results from our experimental evaluation are presented in Section 5. Section 6 offers the conclusion and future work.

2 Related Work

A number of previous studies are related to the present effort. For instance, in [14], the authors present a MapReduce simulator, MRPerf, for facilitating exploration of MapReduce design space. MRPerf captures several aspects of a MapReduce environment, and uses this information to predict application performance. In comparison with an analytical performance model, the simulator captures many more details of the real system, but it takes significantly longer to run.

Ganapathi [5] applied machine learning techniques to predict the performance of MapReduce workloads. The modeling approach consists in correlating the pre-execution characteristics of the workload with measured post-execution performance metrics (e.g. average job response time, resource utilization). However, the authors do not address pipeline parallelism in HOP workloads.

Pavlo [11] compares the performance of MapReduce systems with parallel database systems, finding that MapReduce-based systems were significantly slower when performing a variety of analytic tasks. This finding motivated Jiang et. al. [6] to address the question: must a system sacrifice performance to achieve flexibility and scalability? Their results show that with proper implementation, the performance of MapReduce can be improved by a factor of 2.5 to 3.5, outperforming parallel databases. These previous efforts do not focus on predicting but rather on characterizing the performance of MapReduce, being thus complementary to our work.

Other previous studies used stochastic hierarchical models [9, 8, 7] to predict the performance of parallel applications. In such models, the higher level captures the synchronizations among tasks, expressed by a task graph, whereas the lower level captures the contention at shared resources, represented by a queuing network model. This combination gives a *trade-off* between effectiveness and efficiency for analytical models. Note that if taken separately, these two modeling strategies loose expressivity [7] since queuing network models do not capture synchronization delays among tasks and task graphs do not address delays due to resource contention.

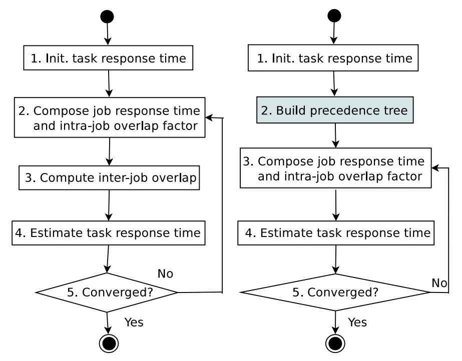
One such solution was proposed by Thomasian and Bay [12]. The precedence among tasks was modeled by a Markov Chain whereas a closed queuing network was used to compute the state transition rate. Although this method captures the synchronization of tasks in a pipeline, the state space grows exponentially with the increase of the number of tasks, making it unpractical to be applied to model jobs with many tasks, as is commonly the case of HOP jobs. Mak and Lundstron [9] proposed an alternative solution with polynomial time that modeled task precedence by a serial-parallel task graph. Liang and Tripathi [8] extended this work to allow task graphs containing other types of operators, although none of them explicitly address pipeline parallelism. We further discuss this solution, here referred to as reference model, next, as it is the basis for our proposed model.

2.1 Reference Model

Liang and Tripathi [8] proposed an analytical model to predict the performance of parallel workloads. Their model combines a precedence tree, which establishes the order of execution of tasks belonging to the same job and defines the synchronization constraints for job execution, and a closed queuing network to address resource contention. In the precedence tree, each leaf node represents a task, and each internal node is an operator (see below) describing the constraints in the execution of the tasks in their children nodes. One example precedence tree is shown in Figure 3.

An approximate Mean Value Analysis (aMVA) technique is used to solve the queuing network. To do so, they introduce the effect of the precedence rules in the aMVA algorithm by inflating the mean queue length (which usually captures only resource contention) by overlap factors representing the overlap in the execution times of tasks belonging to the same job (intra-job overlap) and tasks belonging to different jobs (inter-job overlap). Both overlap factors are estimated based on the precedence tree, given as input to the model.

The authors consider precedence trees built from four types of primitive operators, namely, *serial* (S), *parallel-and* (P_A), *parallel-or* (P_O) and *probabilistic fork* (P_F). An S operator is used to connect tasks that must run in sequence, whereas tasks connected by a P_A run in parallel and must all finish before job execution proceeds. A P_O operator also connects tasks that run in parallel. However,


1. Init. task response time
2. Compose job response time and intra-job overlap factor
3. Compute inter-job overlap
4. Estimate task response time
5. Converged?
No
Yes
1. Init. task response time
2. Build precedence tree
3. Compose job response time and intra-job overlap factor
4. Estimate task response time
5. Converged?
No
Yes

(a) Reference Model [8] (b) Our solution

Figure 1. Main Steps of Analytical Model.

unlike P_A, the group of tasks connected by a P_O finishes execution as soon as one of the tasks finishes. And for P_F only one of the tasks is executed with a probability p.

The main steps of the algorithm proposed by Liang and Tripathi are shown in Figure 1(a). The input parameters are the number K of service centers (physical resources), the service demand matrix $D_{i,k}$, which specifies the average demand of task i in center k, and the multiprogramming level MPL, which defines the average number of concurrent jobs in the system. The algorithm starts by initializing the average residence time of each type of task at each center and the average response time of each task in the system (step 1). Next, the algorithm estimates the average job response time jointly with the intra-job overlap (step 2), followed by the inter-job overlap (step 3). These overlap factors are introduced in the aMVA solution to produce new estimates of task average response time (step 4). Finally a convergence test is performed on the new estimates of average response times. If the new estimates are close enough to the previous ones (i.e., they converge), a final average job response time as well as other performance metrics (throughput, resource utilization) are produced. If the convergence test failure, the algorithm returns to step 2. Parameter ϵ (10^{-4} in our experiments) is used in the convergence test.

The estimation of the average job response time (step 2) is based on a depth-first traversal of the precedence tree. It visit nodes in pre-order, walking from the leaves to the root, recursively, composing the average response time for each internal node according to the operator (S, P_A, etc.) assigned to it. It does so by using that operator to estimate the average response time of the pair of sub-jobs J_1 and J_2 corresponding to the two sub-trees parented by that node. This process continues until it reaches the root, when the average job response time is computed. Due to space constraints, we omit the equations used to perform each step of the algorithm, referring to [8] for a detailed description of the reference model.

Liang and Tripathi focused mainly on how to introduce precedence constraints in the iterative aMVA algorithm. Thus, they assume that the precedence tree is given as input to their model. Moreover, they assume input trees built from basic operators, none of which explicitly capture the pipeline parallelism of HOP workloads.

3 HOP: Architecture and Workloads

In general, the infra-structure of HOP clusters consists of n hosts (nodes), where there is a single master node and several worker nodes. The master node executes the JobTracker process and is responsible for accepting new jobs and assigning them to worker nodes. Each worker node executes a TaskTracker process and manages the execution of the tasks assigned to its node. Each worker node has one or more attached disks with one local file system and one shared, called Hadoop Distributed File System (HDFS).

A HOP job reads one or more input files and produces one output file. Each file is a sequence of registers containing two fields: a key and a value. Each job has two phases: map and reduce. This files are logically partitioned defining the key range to be processed by each worker node. Each partition is divided into sub-partitions of equal size, called splits. Each map and reduce phase is divided, into m and r tasks, respectively. Moreover, the numbers of threads available in each node to execute (in parallel) map and reduce tasks are also fixed, given by parameters pm and pr, respectively. The master node uses a heartbeat protocol to check the status of the worker nodes. When the master node identifies that a worker node has a map/reduce thread idle (i.e., it finished processing its previously assigned map/reduce task), it assigns a new task of the same type to that thread.

Each map task reads registers from one split of the input file. For each split register, it executes the function MAP() and produces one or more output registers. The output registers are then sorted and written in a temporary file in the local file system. At this point the map task terminates and the corresponding thread becomes idle.

Each reduce task is responsible for processing a range of keys produced by the map tasks. One reduce task has three sub-phases: *shuffle*, *sort* and *reduce*. A shuffle sub-phase requests each node to traverse the temporary file produced by the maps, searching for registers in the key range of the reduce which it belongs to. Besides the task level parallelism, HOP allows also that each reduce executes a number ps of shuffle instances in parallel. A shuffle phase is divided into $\frac{m}{ps}$ sub-tasks, where each sub-task processes the results of ps map tasks (in parallel). Thus, each shuffle will be associated with ps map tasks. After each shuffle, a partial sort is performed. The registers are written in an output file in the local file system of the reduce node, which may be different from the node that the shuffle is executing. When all partial sorts finish, a final sort, which merges the temporary files produced by the shuffles, is then performed. Finally, a reduce sub-phase reads this merged file, applies the function REDUCE() and writes the results in the shared file system.

Note that the results produced by each map task should be processed by all reduce tasks. That is, when a map task, running on node n_i, finishes its execution, each reduce task, running on node n_j, sends a shuffle sub-task to n_i to transfer the data from n_i to n_j. Moreover, after the map running on node n_i finishes, a new map task (if there is still one to be executed) is dynamically allocated to n_i. Note that, a reduce task may block if, when all instances of a shuffle sub-task finish to process the results of a sub-set of map tasks, there is no other finished map. Thus, one can see the communication between the collection of map tasks and the set of shuffle sub-tasks of each reduce as a pipeline, where each shuffle (consumer) needs to process the results processed by all maps (producers).

4 Our Analytical Performance Model

This section presents the analytical model developed to predict the performance of HOP workloads. Section 4.1 describes how we extend the reference model [8] to capture the pipeline synchronization. Section 4.2 presents how we model the pipeline parallelism of HOP jobs. The identification of the precedence rules in the timeline as well the estimates of average response time of the internal operators of the tree are presented in Section 4.3.

4.1 Modeling Strategy

We propose an analytical model to estimate the average performance of HOP jobs. Our discussion is focused on average job response time, but we note that other performance metrics, such as throughput and resource utilization, are also computed using Little's Law [10]. Our modeling approach is based on a hierarchical model. The lower level model represents an architecture with n nodes, each with c CPUs and d disks, where disks are connected to the CPUs by a fiber channel and the nodes are inter-connected by a high-speed network. We model this architecture with a closed queuing network with service centers representing each CPU, each disk, the fiber channel and the network. Memory constraints are not explicitly modeled. The higher-level model is given by a precedence tree that captures the synchronization and precedence constraints among tasks.

The workload is composed by a number (MPL) of jobs executing concurrently in the system. Each job has m map tasks and r reduce tasks. Recall that, a partial sort is performed after each shuffle. Since these two phases are performed in sequence, we choose, for modeling purposes, to group each pair of shuffle and sort into a single sub-task named *shuffle-sort* (ss). Similarly, after all partial sorts finish, a final sort, followed by the final phase that applies the reduce function are sequentially executed. We also group this two phases into a single sub-task named *merge*. Thus, in our model, each reduce is composed by $\frac{m}{ps}$ *shuffle-sort* sub-tasks (running ps instances) and one *merge* sub-task. We consider individual sub-tasks as the main workload unit. For that purpose, a map task can be also seen as sub-task. For the sake of clarity, through the rest of this paper, we refer to maps, shuffle-sorts and merges as simply *tasks* belonging to a HOP job.

The main challenge to develop our model is how to capture the synchronization delays introduced by the pipeline parallelism that occurs among maps and shuffle-sorts.Our solution is built from the reference model, briefly described in Section 2.1. Recall that the reference model considers that the precedence tree is given as an input parameter. However the task precedences of HOP jobs are not static, preventing us to give the tree as input. For instance, a shuffle-sort task initiates execution whenever the first map finishes. This moment depends on the resource contention experienced by each map, that is, a map may take longer to finish if it executes concurrently with many other tasks, all of them sharing the same physical resources (e.g., CPU, disks). Moreover, a shuffle-sort may block or not depending on the dynamics of the execution of the other maps. Shuffle-sorts, executing on the same node, may also experience contention at the shared resources and thus, extra delays. The response time of the shuffle-sorts, in turn, determine the time when a merge task initiates. In other words, there is a complex dependency. The precedence tree depends on the response times of the individual tasks, which depend on the contention experienced by them. Resource contention depends on which tasks execute concurrently, which, in turn, depends on the precedence tree.

We address this issue by proposing an algorithm to dynamically build the precedence tree for a HOP job, and add it as an extra step to the algorithm proposed in [8] as our solution is shown in Figure 1(b).

There are two challenges to dynamically build this precedence tree: (1) how to estimate when each task starts and finishes, and (2) how to estimate the average response time of the internal nodes of the tree that will be discussed further in the next sections.

In our current solution, we fix $ps = 1$, although it can be easily extended to allow larger values. Moreover, since, in general, HOP clusters are used to run jobs with massive parallelism, there will not be available resource utilization to run more than one job concurrently. Thus we here assume $MPL = 1$, considering a single job in the system at at time. As consequence, the inter-job overlap, computed in step 3 of the original algorithm (see Figure 1(a)) is skipped in our solution (Figure 1(b)).

4.2 Modeling Pipeline Parallelism

Given that our model provides average estimates of the performance of a HOP job, the precedence tree should capture the synchronization delays and the parallelism experienced, on average, by the job tasks. The strategy adopted to build such tree is to start with a timeline of the execution of individual tasks, which is built based on estimates of each task's average response time and a set of rules defining the precedence constraints among different tasks. In this section, we discuss, in high level, how the precedence tree can be built given this timeline. A detailed discussion on how to build this timeline as well as how we estimate the average response time of the internal operator of the tree are presented in Section 4.3.

Figure 2 shows a simple example of a timeline representing the execution of a HOP job, composed of $m = 2$ maps and $r = 1$ reduce, running on one node ($n = 1$). This node has two threads to process the maps ($pm = 2$) and one thread to execute the reduce ($pr = 1$). The reduce comprises four shuffle-sorts (R_1-R_4) and one merge (R_M) tasks, which must be executed in sequence. To facilitate understanding, we use this simple timeline to illustrate our strategy to build the precedence tree corresponding to it. We note however that the strategy can be applied to much more complex jobs running on larger infra-structures.

As shown in the Figure 2, the two map threads start execution immediately at the beginning of the job execution, whereas the reduce thread remains blocked. As soon as the first map finishes, the first shuffle-sort task (R_1) initiates execution, and another map task M_3 is assigned to the thread that was executing M_1. This point in time, shown in the Figure 2 by a dotted vertical line, marks a synchronization point when the set of tasks executing in parallel changes: M_3, M_2 and R_1 are now in execution. Note that, since M_3 is executed only after M_1 finishes, there is a serial precedence between them. The same holds for R_1 and M_1. After R_1 finishes, the reduce thread blocks, waiting for the next map to finish. Once M_2 finishes, there is another synchronization point marked by the beginning of R_2's execution. The next synchronization point occurs only when M_4 finishes and R_4 starts executing. Note that R_3 can begin execution immediately after R_2 finishes, since M_3 has already finished by this time.

Thus, the execution of a HOP job can be seen as a series of synchronization points, each of which delimits the parallel execution of different sets of tasks. During such "parallel phase", a set of maps and reduce tasks execute in parallel. Since the same map task may execute during multiple parallel phases (e.g., M_2 in Figure 2), we use superscripts to distinguish between multiple phases of the execution of the same task (e.g., M_2^1 and M_2^2) and treat each such phase as an independent (sub-)task. Clearly, these (sub-)tasks must execute sequentially.

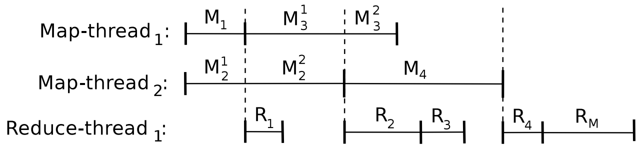


Figure 2. Timeline of a HOP Job.

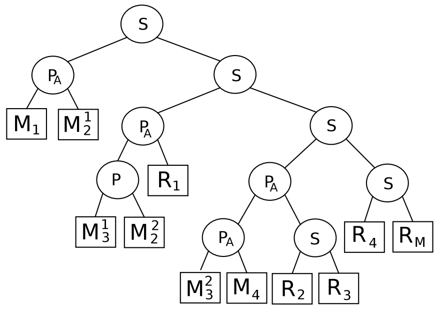


Figure 3. Precedence tree.

Given a timeline, as the one shown in Figure 2. Each leaf node represents a map, shuffle-sort or merge task (M_i and R_i), internal nodes are either S or P_A operators, and the root represents the job. (Sub-)tasks representing phases of the execution of the same task, as well as tasks executed by the same thread must run in sequence (S), whereas tasks executed by different threads during the same phase run in parallel (P_A).

The precedence tree is built by creating multiple subtrees rooted by a P_A operator, one for each parallel phase, and connecting them using S operators. Thus, each P_A-rooted subtree corresponds to a parallel phase. An S node is added to represent the end of a phase, the serialization of maps dynamically allocated to the same thread, and the serial execution of last shuffle-sort and the merge tasks of a given reduce. Figure 3 shows the precedence tree corresponding to the timeline shown in Figure 2.

Generally speaking, the tree is built by mapping each synchronization point into an S-rooted subtree with two children: the left child is a sub-tree representing the previous phase and the right child is, if is not the last phase, an S-operator for linking with other phase sub-trees, or, the next phase sub-tree. Each P_A-rooted sub-tree of a phase has two branches (i.e., childresn): one for each parallel map threads, in left side, and one for parallel reduce threads, in right side. is a P_A-rooted subtree with two branches (i.e., children): one for the parallel map threads and the other for the parallel reduce threads. Each such branch may thus be a P_A-rooted subtree itself representing the parallel execution of multiple threads of a given type. S nodes may be added to represent the serial execution of multiple tasks of the same type (map or shuffle-sort) by the same thread during the same parallel phase.

Note that the exact shape of the precedence tree depends on the number of synchronization points, which in turn, depends on the average response time of each task. Thus, it must be rebuilt at each iteration of the algorithm (Figure 1(b)) using the average task response times estimated in the previous iteration. Once the tree is built, it can be used along with the Liang and Tripathi solution to produce new estimates of task and job of job performance. We do that by considering individual tasks (and, in the case of tasks spanning multiple phases, individual sub-tasks) as the main workload unit. In the following, we discuss how we build the timeline of a HOP job execution given the (current) estimates of (sub-)task average response times. We also discuss how we estimate the average response time corresponding to a parallel phase of the job execution.

4.3 Precedence rules and Response Time Estimates

To define the execution timeline of a HOP job, we must identify events of termination of (sub-)tasks. Such events may trigger synchronization points and the beginning of the execution of other (sub-)tasks. That is, at each such event, a set of precedence rules, defined based on the pipeline among maps and shuffle-sorts and on the serial relationships among (sub-)tasks (e.g., maps dynamically allocated of to the same thread), are applied to determine whether new tasks start execution and whether there is blocking. To capture the task synchronization of the pipeline, we assume that each reduce thread, which executes all shuffle-sorts and the merge of a single reduce, has a logical queue associated to it. Since each map is processed by all reduces, each map, after finishing execution, puts a "token" in the logical queue of each reduce to indicate that its results are ready to be processed by them. A shuffle-sort only starts execution if there is at least one token to be consumed at the logical queue of its corresponding reduce thread. The following events are considered:

Map M_i is the next to finish: this event occurs if the cumulative average response time of the thread executing M_i, defined as the sum of the average response times of all already map tasks executed by M_i's thread is the smallest among the cumulative average response times of all executing map and reduce threads. If this event occurs, we insert a token corresponding to this map in the logical queues of all reduces. We must then check whether any reduce thread is blocked. If so, it must be unblocked, and new shuffle-sorts start executing by the previously blocked reduce threads. The token is removed from the logical queue when the shuffle-sort starts. Moreover, if there are still maps waiting to execute, one of them is dynamically allocated to M_i's thread, which starts executing it. Note that, if there are blocked reduce threads, this event implies in the end of a parallel phase (i.e., synchronization point).

Shuffle-sort S_i is the next to finish: if S_i is the last shuffle-sort to be executed by its reduce thread, then the merge task starts execution. Otherwise, we check the logical queue. If it is not empty, then the next shuffle-sort (S_{i+1}) starts execution and removes one token from its logical queue. Otherwise, if the logical queue is empty, the reduce thread blocks waiting for a map to finish.

Merge R_{Mi} is the next to finish: in this case the corresponding reduce thread terminates its execution.

The basic idea of the algorithm is to dynamically progress in the execution of the job by comparing the cumulative average response times of all map and reduce threads and defining events of termination of tasks. At each such event, we apply the rules described above to move forward in the timeline. At the same time, we add new nodes and sub-trees to the precedence tree corresponding to new (sub-)tasks and phases added to the timeline, as described in Section 4.2.

Our algorithm takes as inputs the average response time of each task (M_i, R_i and R_{M_i}), the number of threads per type of task (pm and pr), and the total number of tasks (C). The latter can be computed from the numbers of map tasks (m) and reduce tasks (r). The algorithm builds the tree from top to bottom as it progresses in the timeline.

Once we have built the tree, the next challenge is how to estimate the average response time associated with an internal P_A-node in the precedence tree. Note that this corresponds to estimating the average of the distribution of the maximum of k random variables, which is not the same as taking the maximum of the averages of the variables. Indeed, we did verify, in the preliminary experiments, that using the latter leads to great approximation errors. Thus, we here consider two approaches:

- **Tripathi-based**: following [8], we approximate the response time distribution associated with a P_A-rooted subtree to an Erlang or hiper-exponential distribution according with its CVs and take the average of the selected distribution.
- **Fork/join-based**: we consider the execution of a parallel-phase as a fork-join block, and use previously adopted estimates of the average response time of fork/joins [13]. One such estimate is the product of the k-th harmonic number ($\sum_{i=1}^{k} \frac{1}{i}$) by the maximum average response time of k tasks.

For both approximations, we assume that task response times are exponentially distributed, as in [8]. As a final note, recall that in a real system, the first shuffle-sort starts when the first map finishes. Thus, the smallest response time of all running maps defines when the shuffle-sort starts. Since we build the timeline and the precedence tree based on average response times, in order to estimate the starting time of a shuffle-sort, we must take the average of the minimum of the response times of all running maps. This corresponds to taking the average of the minimum of different exponential distributions. Once again, this is not the same as taking the minimum of the averages.

5 Experimental Analysis

We developed an event-driven queuing network simulator to validate our model, and particularly, its assumptions (e.g., exponentially distributed task response times). Our simulator reproduces the dynamics of HOP job execution in terms of contention for resources and synchronization constraints. Unlike the analytical model, which is based on average estimates, the simulator reproduces the execution of individual tasks. That is, it captures dynamic aspects of the workload (e.g., individual blocking events, burst arrivals, etc.), which are only approximated by our analytical model.

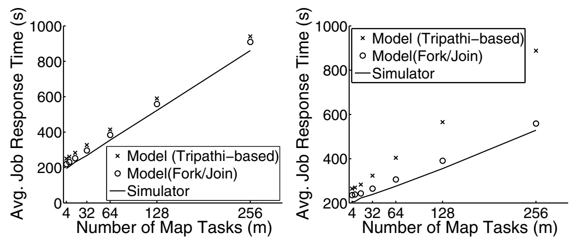
For both simulation and model, we assume an architecture consisting of four nodes ($n = 4$) with four CPUs ($c = 4$) and one disk ($d = 1$). The workload is composed of a single HOP job ($MPL = 1$), containing m map tasks and 4 reduce tasks. We experimented with m varying from 4 to 256. The maps are dynamically allocated among the 4 nodes. Each reduce runs on one node. We assume that the HOP job reads one input file, that is uniformly partitioned among all four nodes. Average service demands for each type of task on each modeled resource are shown in Table 1. These are input to both model and simulator. In particular, for the simulator, we assume that task residence times at each center are exponentially distributed with averages given by Table 1. We note that these demands are representative of a real setup, i.e., they were collected from a real HOP platform. We also note that the network is used only by the shuffle-sort task to transfer the data produced by a map from its original node to the node where the corresponding reduce thread is running. If both run on the same node, network demand is zero.

pm	Task	CPU	Fiber	Disk	Network
1	Map	5,0764	0.0677	3.1844	0.0000
	SS	0.5450	0.0085	0.4003	0.1815
	Merge	39.9561	1.2583	59.2157	0.0000
4	Map	5,1057	0.0677	3.1844	0.0000
	SS	0.6307	0.0085	0.4007	0.1817
	Merge	45.8190	1.258	59.2157	0.0000

Table 1. Mean service demands.

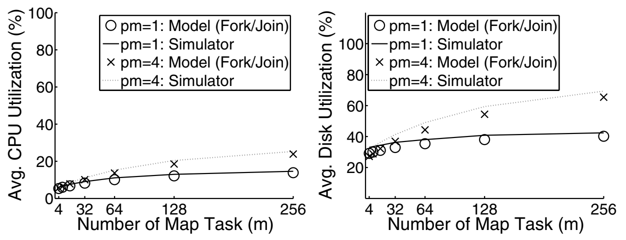
The simulation results discussed below are averages of 5,000 replications. Confidence intervals show that results from individual replications deviate from averages by at most 1.04%, with 95% confidence. They are omitted from figures to improve readability.

We start by fixing the number of nodes and considering two scenarios, namely with and without parallelism among map tasks. We do so by considering $pm = 4$ and $pm = 1$, respectively. Figure 4 presents the average job



(a) $pm = 1$ (b) $pm = 4$

Figure 4. Mean job response time validation.



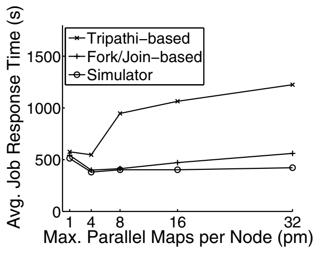
(a) CPU (b) Disk

Figure 5. Mean resource utilization validation.

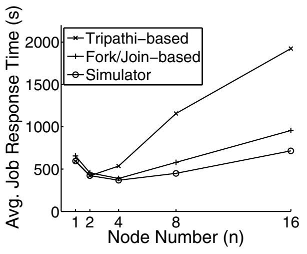
response time, estimated by our model and by the simulator, as the number of map tasks increases for both $pm = 1$ and $pm = 4$. For the model, we show results for the two approaches used to estimate the average response time of a parallel phase.

Starting with the $pm = 1$ scenario, we find that the maximum relative difference between analytical model and simulation was 29%, when we estimate the response time of parallel phases using the Erlang approximation proposed by Tripathi and Liang [8]. However, the use of the fork/join approximation leads to much better results: the maximum relative difference is below 13%. Both approaches also present good agreement with the simulator in terms of resource utilization. Figure 5 shows results for CPU and disk utilization, for simulator and the fork/join-based model. Results for the Tripathi-based model are omitted due to space constraints. The maximum relative difference in resource utilization is for disk utilization, falling below 22% for the Tripathi-based estimate and 12% for the fork/join.

The average response times of individual tasks were also reasonably well estimated, with a maximum relative difference below 13% and 7%, for maps and merges, respectively, considering the fork/join approximation. The average response times of shuffle-sort tasks, on the other hand, were underestimated (relative difference as high as 43%), due to underestimated fiber and network utilizations. However, given that in our considered setup, which is realistic of a real setup, shuffle-sort tasks demand much fewer resources than the other two types of tasks, the impact of such underestimate on the job average response time was marginal.



(a) Impact of pm ($n = 4$)



(b) Impact of n (m=128, $pm = 4$)

Figure 6. Impact of System Parameters.

Nevertheless, we intend in the future to further investigate how to improve our model to more accurately estimate the shuffle-sort average response time.

For scenario $pm = 4$, map tasks face more contention for shared resources, which imply in queuing delays. The fork/join-based model shows a good agreement with simulator, with a maximum relative difference under 15%, whereas the Tripathi-based approach over-estimated job average response time by as much as 68%. Thus, we find that the Fork-Join estimate provides an improvement, quite significant in some scenarios, over the approach proposed by Tripathi and Liang and explored in the reference model.

We extend our validation by evaluating the impact of the number of map threads (pm) and number of nodes (n) on the estimated job response time. Once again,we compare analytical versus simulation results. We start by fixing the number of nodes $n = 4$ and the number of map tasks $m = 128$ and varying pm. For $pm = 1$ the map tasks are serially executed in each node, while for $pm = 32$ there may be 32 map tasks in parallel per node. While increasing the number of map tasks, we kept fixed the total demand for each resource. Figure 6(a) shows that, once again, the Tripathi-based model over-estimated mean job response time whereas the fork/join-based estimate presents a much better agreement with the simulator, with a relative difference typically under 22%. Since each node has four CPUs, when $pm = 1$ the CPUs were underutilized, presenting less contention, as we can see a decay in the beginning of the curve in Figure 6.

Figure 6(b) results for varying number of nodes. For these experiments, we fixed $m = 128$ and $pm = 4$. Note that, as we increase the number of nodes, we also increase the number of reduces, keeping one reduce per node, and ultimately leading to an increase in the job average response time. The Tripathi-based model, as in the previous experiments, over-estimates the simulator for values of $n \geq 4$ (as much as 45%) whereas the fork/join estimate provides a much closer approximation, with maximum relative difference under 34%.

6 Conclusion and future work

In this paper, we addressed the challenge of modeling HOP workloads, which exhibit intra-job pipeline parallelism. The modeling approach extends the solution proposed in [8] which is based on a hierarchical model where the HOP execution flow is represented by a precedence tree and the contention at the physical resources are captured by a closed queuing network. Our main contribution is a method to build a precedence tree for HOP jobs that captures the pipeline synchronizations. We also propose the use of a tighter and more accurate approach to estimate the average response time of parallel phases of the job execution. We validate our model against a queuing network simulator, where our performance model presents a close agreement, with maximum relative difference under 15%. As future works we plan to validate our model against results obtained from a real system, extend our model to address other types of resources (e.g., main memory) and to investigate the impact of parameter ps, the maximum number of shuffle running in parallel per reduce, on system performance.

7 Acknowledgements

This work was partially supported by the INCT-Web (MCT/CNPq grant 57.3871/2008-6), and by the authors scholarships from CNPq, FAPEMIG and CAPES.

References

[1] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *USENIX*, 2010.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Op. Systems Design & Implementation*, 2004.

[3] Apache Software Foundation. Powered by hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[4] Apache Software Foundation. Official apache hadoop website. http://hadoop.apache.org/, Jun, 2011.

[5] A. Ganapathi. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning*. PhD thesis, University of California, 2009.

[6] D.R. Jiang, B. Ooi B., L. Shi, and S. Wu. The performance of mapreduce: an in-depth study. *VLDB*, 3:472, 2010.

[7] H. Jonkers. Queueing models of parallel applications: the Glamis methodology. In *Comp. Perf. Eval.*, pages 123–138. Springer, 1994.

[8] D. Liang and S. Tripathi. On performance prediction of parallel computations with precedent constraints. *IEEE Trans. Parallel Distrib. Syst.*, 11:491–508, 2000.

[9] V. W. Mak and S. F. Lundstrom. Predicting performance of parallel computations. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):257, 1990.

[10] D. Menasce, W. Dowdy, and V. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, 2004.

[11] A. Pavlo., E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[12] A. Thomasian and P. F. Bay. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Trans. Comput.*, 35(12):1045–1054, 1986.

[13] E. Varki. Mean value technique for closed fork-join networks. *SIGMETRICS Perform. Eval. Rev.*, 27:103–112, May 1999.

[14] G. Wang, A. Butt, P. Pandey, and K. Gupta. Using realistic simulation for performance analysis of MapReduce setups. LSAP. ACM, 2009.