On Task Assignment in Data Intensive Scalable Computing

Giovanni Agosta, Gerardo Pelosi, and Ettore Speziale

Dipartimento di Elettronica, Informazione e Bioingegneria – (DEIB)
Politecnico di Milano,
Piazza Leonardo da Vinci, 32, I-20133, Milano, Italy
`name.surname@polimi.it`

Abstract. MapReduce and other Data-Intensive Scalable Computing paradigms have emerged as the most popular solution for processing massive data sets, a crucial task in surviving the "Data Deluge". Recent works have shown that maintaining data locality is paramount to achieve high performance in such paradigms. To this end, suitable task assignment algorithms are needed. Current solutions use round-robin task assignment policies, which was shown to yield suboptimal results. In this paper, we propose and evaluate new algorithms for task assignment on a model of the Hadoop framework, comparing them with state-of-the-art solutions proposed in theoretical works as well as with the current Hadoop task assignment policies.

1 Introduction

The data-intensive computing paradigm has recently received significant attention in both research and industrial ICT communities due to the exponential increase of data available for analytical processing–the so-called "Data Deluge" [7]. The cloud computing scenario represents the most important arena where the potential impact and the effectiveness of data-intensive computing are most visible. The Cloud is an abstraction for the complex infrastructure underlying the Internet and refers to both the applications delivered as services over the network and the hardware and software resources that provide those services. As a key concept, the cloud computing paradigm shifts data storage and computing power away from the user endpoints, across the network, and into large clusters of machines hosted by cloud providers (e.g., Amazon, Google). The research challenges aimed at exploiting the full potential of data-intensive computing lie in designing clusters and software frameworks to improve performance of massive simultaneous computations, energy efficiency, and reliability of the provided services. In this regard, *MapReduce* is the leading software framework, composed of both a programming model and an associated run-time system, introduced by Google in 2004 to support distributed computing on large data sets, through splitting the workload over large clusters of commodity PCs [4, 5]. A critical issue to achieve good performance on large scale *MapReduce* systems lies in ensuring that as many data accesses as possible are executed locally. To this end, a data processing job is parallelized in a set of tasks, which are assigned to servers which will execute them. However, purely locality-based scheduling may lead to long latencies, since a specific computation may access data stored on busy servers. Thus, locality-aware, latency minimizing scheduling algorithms have been designed [6] to reduce latency while still exploiting locality.

Problem Settings and Contributions We present an algorithm for task assignment on a cluster of servers that balances latency and resource usage, while also taking into account the workload running on the target cluster. The proposed algorithm is able to achieve an efficient tradeoff between latency and resource usage through employing a novel heuristic technique. A simulation-based analysis of the performance of the proposed algorithm against the state-of-the-art solutions is presented, showing that it is able to obtain lower latencies than the standard locality aware round-robin strategy [15], as well as lower resource consumption than the flow-based algorithm reported in [6] together with a better computational complexity. Moreover, we show that our algorithm and the flow-based one are Pareto-optimal with respect to latency and resource consumption, while the round-robin is not. On the other hand, the present work does not deal with fault tolerance in MapReduce systems. While this is also a critical issue in achieving performances, it is a different issue from load balancing, which is best covered with specialized approaches that act during the task execution rather than at task assignment. We also do not deal with job scheduling, and therefore with fair-share scheduling among users, as this goal is better achieved at the level of job scheduling.

Structure of the Paper The remainder of the paper is organized as follows. Section 2 reports a brief summary of background information about MapReduce systems. Section 3 defines the abstract model on which the proposed task assignment algorithm is designed. An operative description of the algorithm as well as the description of its properties are also reported. Section 4 presents the evaluation of the proposed algorithm, in comparison with existing practices and theoretical works. Section 5 provides an overview of closely related works, and Section 6 draws some conclusions and highlights future directions.

2 Background

A *MapReduce* system is a framework for distributed computation over large data sets that implements both the MapReduce programming model and an associated run-time system. It mimics the functional programming constructs *map* and *reduce* and enables the programmer to abstract from common distributed programming issues such as: load balancing, network performances and fault-tolerance. In spite of its simplicity, the MapReduce programming model turns out to effectively fit many problems encountered in the practice of processing large data sets although a preliminary decomposition of the problem into multiple MapReduce jobs is often needed [2,9]. Typical applications are Web indexing, report generation, click-log file analysis, financial analysis, data mining, machine learning, bioinformatics and scientific simulations [4,5]. The programming model is based on the iteration over data-independent inputs where the required operations are: *i*) computation of key/value pairs from each piece of input (*map* phase); *ii)* grouping of all intermediate values by the key value; *iii)* reduction of each data group to a few computed values (*reduce* phase). Word counting is a toy example that considers a set of text documents as input and a list of the occurrences of each word as output, where the key/value pair is given by "word"/"counting" instances.

Actual implementations of both proprietary [4, 5] and open-source [15] instances of a *MapReduce* system employ dedicated clusters of commodity machines. Each cluster is managed by a *master* server that is in charge of keeping track of all jobs while they are queued and processed in the distributed system. A *job-tracker* running on the master server schedules the received jobs and assigns their tasks on target *slave* servers. Each slave server runs a *task-tracker* that schedules the corresponding tasks, on a first-come/first-served strategy, consistently with the local computational resources and operating system policies. Due to the simplicity of the MapReduce programming model, a user will seldom submit a single job, since, the composition of more jobs in complex workloads (or applications) allows to take better advantage of the system. A MapReduce application is, in general, a Directed Acyclic Graph (DAG) where the nodes represent jobs and the arcs represent data dependences [2]. Therefore a job can only be executed after all of its predecessors have been completed.

Canonical solutions to the scheduling of a DAG solve a constrained optimization problem where the figure of merit is the expected latency of every job and the constraints are represented by the available resources. A variant of this setting is to employ the minimization of resources as a figure of merit, and the maximum latency allowed for each job as a constraint. However, these strategies cannot be applied in the job-tracker, because they need a precise knowledge of the foreseen latency of each job as well as the available resources. The latency of a MapReduce job is not trivial to predict. This is due to both the heterogeneity of applications submitted by different users, and to the presence of straggled tasks and execution failures, which can change unpredictably the actual latency of the executed job [10]. In addition, the submission rate of the jobs in a Data Intensive Scalable Computing (DISC) cluster is quite low — on average, one job per 2-3 minutes [3, 10] — and thus the time to fill a queue of jobs to schedule is high. Given the aforementioned considerations, the scheduling strategy for the job-tracker of a MapReduce system should take into account the cluster workload variation over time. Therefore, *online* scheduling algorithms represent the prime choice. Indeed, proprietary and open-source MapReduce systems adopt online scheduling strategies. Apache Hadoop [1] is an open-source Java implementation of *MapReduce*, originally designed to implement parallel processing in local networks, whose job-tracker employs a round-robin strategy (over the available resources) to assign the tasks in each job over the slave servers. A more accurate task assignment algorithm is proposed in [6], where the authors describe a flow-based algorithm aimed at minimizing the completion time of the considered job and show how such solution is near-optimal within an additive constant from the optimum solution obtained through the fully combinatorial exploration of task assignments. We extend the abstract system model presented in [6], to effectively obtain a trade-off between job latency and throughput. Moreover, through taking into account a pre-existing workload, we better represent the challenges of an on-line task assignment.

3 A Locality Aware and Bounded Latency Approach

In this section, we introduce the main contribution of this work, a *Locality Aware Bounded Latency* (LABL) task assignment algorithm. We will now provide some preliminary concepts and definitions, followed by a description of the algorithm. We describe the formal properties of the LABL task assignment algorithm, and show that its running time complexity is linear w.r.t. the size of the input job.

3.1 Preliminaries

Definition 1. *A* job *is a set of* tasks*, $T=\{t_1,\ldots,t_m\}$. The tasks are mutually independent and do not have any control or data dependencies among them. Thus, the job can be fully parallelized.*

In a *MapReduce* implementation, the tasks are partitioned between *map* and *reduce* operations. The *reduce* tasks must be scheduled after the *map* tasks have completed [15]. Without loss of generality, it is safe to model jobs as composed only of *reduce* tasks or only of *map* tasks. A job composed of both types of tasks is split in two homogeneous jobs for the purpose of the model, with the provision that the *reduce* job is scheduled only after the corresponding *map* job has completed. Note also that, in practice, the distribution of latencies of *reduce* tasks is remarkably similar to that of *map* tasks [10], so it is not necessary to keep track of *map* and *reduce* jobs separately.

Definition 2. *A* cluster *is a set of homogeneous* servers*, $S=\{s_1,\ldots,s_n\}$, each of which is assumed to be able to execute a given task with the same execution time, provided that a copy of the corresponding data is locally accessible.*

The locality of the data processed as the input of each task is crucial for the performance of the whole system. Indeed, the overall performance in terms of both job latency and total system workload largely depends on the initial data placement on the cluster.

Definition 3. *Given a job T and a cluster S, a* data placement function *ρ specifies the subset of servers where the execution of a task t can be completed through accessing a local copy of the necessary data.*

$$\rho : T \mapsto 2^S \text{ and } \forall\, t \in T,\ \rho(t) \subseteq S$$

The number of data copies available for a given task $t{\in}T$ is denoted as $|\rho(t)|$. A task t is denoted as local *to a server s if $s \in \rho(t)$, and as* remote *otherwise.*

As previously mentioned, the considered abstract model assumes a set of homogeneous tasks and a set of homogeneous servers, in such a way that all the tasks which data is locally available run in the same amount of time (w_{loc}) and all tasks running on servers where remote data accesses must be employed also exhibit the same execution time (w_{rem}). The execution time experienced by the latter type of tasks depends on the total number of remote data accesses observed in the system. However, the additional overhead (with respect to the execution time of a task accessing data in place) does not incur in large variations when the network traffic of the system is in a steady state [6]. Therefore, the usual conservative assumption about the execution time experienced by tasks accessing remote data (fitting most of the practical environments) considers these execution times constant (over the entire set of tasks). In particular, the execution times are three times higher than the ones of tasks accessing data in place [6,11].

Definition 4. *Given a job T and a cluster S, an* assignment *corresponds to the execution of a number of tasks $\{t_1, \dots, \} \subseteq T$ on a single server $s \in S$, and is denoted as a pair $(s, \{t_1, \dots, \})$. A* Task Assignment*, $\mathcal{A}$, is a collection of pairs (s', T') with $s' \in S$, $T' \subseteq T$, such that every task in T and every server in S is present in one and only one assignment.*

$$\mathcal{A} = \begin{cases} (s', T') : s' \in S, T' \subseteq T \\ \forall s'' \in S, T'' \subseteq T \nexists (s'', T'') : s'' = s' \vee T'' = T' \end{cases}$$

The assignment of tasks to servers dynamically influences the subsequent assignment choices, due to the potential change of both network traffic and workload level of the cluster. The *job-tracker*, running on the *master* server, is the system actor in charge of orchestrating the workload distribution thus, it can dynamically evaluate the *load* of each server. Assuming w_{loc} and w_{rem} as the unitary task execution times for processing local and remote data, respectively, the evaluation of any server load is abstracted through the definition of the following function. We call the time w_{loc} a *unit of work*.

Definition 5. *Let T be a job, S be a cluster, and $\mathcal{A}$ a given task assignment. The* load *of any server $s \in S$ is evaluated through a function, ϕ, which maps s to the numerical value of its current workload (measured in* units of work*). The workload of s in assignment $\mathcal{A}$ includes the set of tasks $\widehat{T} \subseteq T$, such that $(s, \widehat{T}) \in \mathcal{A}$. Then,*

$$\phi(s) = \phi_s + w_{loc}|\widehat{T}_{loc}| + w_{rem}|\widehat{T}_{rem}|$$

where $\widehat{T}_{loc} = \{t \in \widehat{T} : s \in \rho(t)\}$ and $\widehat{T}_{rem} = \{t \in \widehat{T} : s \notin \rho(t)\}$ denote the sets of task that access data to be processed locally or remotely, respectively, while ϕ_s is a constant factor that takes into account the load due to the tasks that are already running on s before the assignment $(s, \widehat{T})$ is put into effect.

Note that, without loss of generality, we consider that at least one server s_0 has an initial workload $\phi_{s_0} = 0$, i.e. there is at least one free server. To understand the rationale of this choice, consider a load ϕ for a given cluster S, leading to an assignment $\mathcal{A}$. Now, consider a second load ϕ' such that $\forall s \in S, \phi'_s = \phi_s + 1$.

The same assignment is generated under this second workload, except that the starting time of each task is increased by one unit of time. Thus, to provide a uniform scale for latency measurements, we normalize ϕ so that the condition $\exists s_0 \in S \mid \phi_{s_0} = 0$ holds.

3.2 Optimization Goals

Given a job T and a cluster S, the proposed task assignment strategy aims at achieving a tradeoff between the *job latency* and the total *resource accounting* of the target cluster. The figures of merit used to evaluate the effectiveness of a task assignment algorithm `alg` and the resulting Task Assignment $\mathcal{A}$ are:

(i) The *resource accounting* is defined as the total number $C_{\texttt{alg}}(T)$ of units of work consumed to execute the job:

$$C_{\texttt{alg}}(T) = \sum_{s \in S} (\phi(s) - \phi_s)$$

(ii) The *latency* $l_{\texttt{alg}}(T)$ is defined as the maximum completion time for a task of the job, normalized to the minimum starting time for a task:

$$l_{\texttt{alg}}(T) = \max_{s \in S} \phi(s)$$

(iii) The *throughput* is defined as the ratio $R_{\texttt{alg}}(T)$ between the number of tasks in the job and its resource accounting:

$$R_{\texttt{alg}}(T) = \frac{|T|}{C_{\texttt{alg}}(T)}$$

3.3 Lower Bounds for the Expected Job Latency

We start from the insight that it is possible to drive the online task assignment procedure taking as a reference a lower bound on the job latency. Such a reference allows the assignment procedure to start with a predetermined minimum job latency limit, discarding unfeasible scenarios a-priori and taking into account remote assignments that would not be considered under lower latency limits. Given a job T and a idle cluster S (i.e. $\forall\, s{\in}S, \phi_s{=}0$), if each task can access the data to be processed on every server locally (i.e., $\forall\, t{\in}T$, $\rho(t){=}S$), then a trivial lower bound for the job latency is given by $\lceil\, w_{\text{loc}}|T|/|S|\,\rceil$. Weakening these assumptions through removing either the hypothesis that each server is initially idle or the hypothesis of a uniform placement of data for each task, leads to solve two simpler problems prior to apply any task assignment operation. These problems are more formally stated as follows.

Problem 1. Let S be a cluster with initial workload defined as $\phi(s){=}\phi_s$, $\forall\, s{\in}S$, and T be a set of tasks that can locally access the data to be processed on any server S: $\forall\, t{\in}T$ $\rho(t){=}S$.

Considering the execution cost of each task as w_{loc} (i.e. ignoring the impact of the data placement), a lower bound for the job latency is computed a-priori as:

$$l^* = \left\lceil \frac{w_{\text{loc}}|T|}{|S|} + \frac{1}{|S|} \sum_{s \in S} \phi(s) \right\rceil$$

The straightforward solution of Problem 1 follows from considering each task as a local one since the data is assumed to be uniformly replicated on each server.

Problem 2. Let S be a cluster with initial workload defined as $\phi(s) = \phi_s, \forall\, s \in S$, and T be a set of tasks whose data is replicated on servers according to a data placement function $\rho : T \mapsto 2^S$. Assuming a limit l for the expected job latency, the set S can be partitioned as $S{=}S_{\text{inf}}[l] \cup S_{\text{sup}}[l] \cup S_{\text{busy}}[l]$, where $S_{\text{sup}}[l]{=}\{s \in S | l - \phi_s \geq w_{\text{rem}}\}$ is the set of servers that can only execute local tasks within the latency limit l, $S_{\text{busy}}[l]{=}\{s \in S | l{-}\phi_s{\leq}0\}$ is the set of servers that are busy with workload from previous jobs. Finally, the set of servers that cannot execute remote tasks within the latency limit l is $S_{\text{inf}}[l]{=}S \setminus (S_{\text{inf}}[l] \cup S_{\text{busy}}[l])$. The set of tasks T can also be partitioned as $T = T_{\text{loc}}[l] \cup T_{\text{rem}}[l]$, where $T_{\text{rem}}[l] = \{t \in T | \rho(t) \subseteq S_{\text{busy}}[l]\}$ is the set of tasks that can only be executed remotely within l and $T_{\text{loc}}[l] = T \setminus T_{\text{rem}}[l]$ is the set of tasks that can be run within l on servers with local access to data.

Considering the execution time of any task in T_{loc} as w_{loc} and the execution time of any task in T_{rem} as w_{rem} ($> w_{\text{loc}}$), a lower bound for the expected job latency is derived as:

$$l^{**} = \min_{l \geq 0} \begin{cases} \displaystyle\sum_{s \in S_{\text{sup}}} \left\lfloor \frac{l - \phi_s}{w_{\text{rem}}} \right\rfloor \geq |T_{\text{rem}}[l]| \\ \displaystyle\sum_{s \in S_{\text{sup}} \cup S_{\text{inf}}} (l - \phi_s) \geq a[l] \end{cases}$$

where $a[l]$ is the cost of the execution of the given job following an "ideal" assignment of both local and remote tasks within the latency limit l (in this way, the data placement function is employed only for partitioning the job in the local $T_{\text{loc}}[l]$ and remote $T_{\text{rem}}[l]$ task sets but not to solve assignment conflicts, if any):

$$a[l] = w_{\text{rem}}|T_{\text{rem}}[l]| + w_{\text{loc}}|T_{\text{loc}}[l]|$$

The first inequality states that the servers in S_{sup} can provide, as a whole, enough units of work to manage the execution of all remote tasks within the latency limit of l, while the second inequality constraints the available number of units of work on the entire cluster to be greater than the resource allocation needed to schedule each local task locally and each remote task remotely assuming no resource conflict. Therefore, the minimum among the aforementioned latency limits gives a lower bound l^{**} which guarantee a more accurate estimate with respect to the previous bound l^*, thus allowing to initialize our on-line assignment algorithm with a threshold that guarantee a faster convergence.

```
Algorithm: TaskAssignment
Input: $S = \{s_1, \ldots, s_m\}$, set of servers
       $T = \{t_1, \ldots, t_n\}$, set of tasks
       $l$, initial server load limit
Output: $A = \{(s, \widehat{T}) : s \in S, \widehat{T} \in \wp(T);$
        $\forall (s', \widehat{T}'), (s'', \widehat{T}''), s' \neq s'' \wedge \widehat{T}' \cap \widehat{T}'' = \emptyset\}$,
        set of assignments
// Place tasks on servers through trading off the job latency and data movement
$A \leftarrow \emptyset$
while $T \neq \emptyset$ do
    $S_{\text{inf}} \leftarrow \{s \in S, l - w_{\text{rem}} < \phi(s) < l\}$
    $S_{\text{sup}} \leftarrow \{s \in S, 0 \leq \phi(s) \leq l - w_{\text{rem}}\}$
    $T_{\text{loc}} \leftarrow \{t \in T, \rho(t) \cap \{S_{\text{inf}} \cup S_{\text{sup}}\} \neq \emptyset\}$
    $T_{\text{rem}} \leftarrow \{t \in T, \rho(t) \cap (S \setminus \{S_{\text{inf}} \cup S_{\text{sup}}\}) = \emptyset\}$
    // Phase I
    // Place most constrained tasks in $T_{\text{loc}}$ on most loaded servers
    // unable to execute a remote task while limiting their load
    // under $l$ (i.e. servers in $S_{\text{inf}}$)
    // $T_{\text{loc}} \cup T_{\text{rem}} = T$
    while $S_{\text{inf}} \neq \emptyset$ do
        $s \leftarrow$ ExtractMostLoadedSrv ($S_{\text{inf}}$) // Get $s \in S$ s.t. $\phi(s) > \phi(s_i), \forall s_i \in S, s_i \neq s$
        $\widetilde{T} \leftarrow \rho^{-1}(s)$ // Set containing tasks working on $s$ local data
        $\widehat{T} \leftarrow \emptyset$ // Set of tasks foreseen to be assigned to $s$
        while $\widetilde{T} \neq \emptyset$ and $\phi(s) \leq l$ do
            $t \leftarrow$ ExtractMostConstrainedTask ($\widetilde{T}$) // Get $t \in \widetilde{T}$ s.t. $|\rho(t)| < |\rho(t_i)|, \forall t_i \in \widetilde{T}, t_i \neq t$
            $\widehat{T} \leftarrow \widehat{T} \cup \{t\}$
        $A \leftarrow A \cup \{(s, \widehat{T})\}$
        $T_{\text{loc}} \leftarrow T_{\text{loc}} \setminus \widehat{T}$
    // Phase II
    // Place remote tasks on servers $s$ having a load
    // such that $l - \phi(s) \geq w_{\text{rem}}$
    if ConsiderRemoteAssignments ($l$) = $true$ then
        $S'_{\text{sup}} \leftarrow \emptyset$
        $\widehat{A} \leftarrow \emptyset$
        while $T_{\text{rem}} \neq \emptyset \wedge S_{\text{sup}} \neq \emptyset$ do
            $t \leftarrow$ ExtractTask ($T_{\text{rem}}$)
            $s \leftarrow$ ExtractSrv ($S_{\text{sup}}$)
            $\widehat{T} \leftarrow$ ExtractAssignment ($\widehat{A}$, $s$)
            $\widehat{T} \leftarrow \widehat{T} \cup \{t\}$
            $\widehat{A} \leftarrow \widehat{A} \cup \{(s, \widehat{T})\}$
            if $\phi(s) \leq l - (w_{\text{rem}})$ then
                $S'_{\text{sup}} \leftarrow S'_{\text{sup}} \cup \{s\}$
            if $S_{\text{sup}} = \emptyset$ then
                $S_{\text{sup}} \leftarrow S'_{\text{sup}}$; $S'_{\text{sup}} \leftarrow \emptyset$
        $A \leftarrow A \cup \widehat{A}$
    // Phase III
    // Place tasks on less loaded servers storing the corresponding data
    $T \leftarrow T_{\text{loc}} \cup T_{\text{rem}}$
    $\widetilde{T} \leftarrow \emptyset$
    while $T \neq \emptyset$ do
        $t \leftarrow$ ExtractTask ($T$)
        $s \leftarrow$ ExtractLeastLoadedSrv ($\rho(t)$)
        if $\phi(s) + w_{\text{loc}} \leq l$ then
            $\widehat{T} \leftarrow$ ExtractAssignment ($A$, $s$)
            $\widehat{T} \leftarrow \widehat{T} \cup \{t\}$; $A \leftarrow A \cup \{(s, \widehat{T})\}$
        else
            $\widetilde{T} \leftarrow \widetilde{T} \cup \{t\}$
    $T \leftarrow \widetilde{T}$
    $l \leftarrow l + 1$
return $A$
```

Fig. 1. Locality Aware & Bounded Latency (LABL) Task Assignment Algorithm

3.4 Task Assignment Algorithm

The LABL Task Assignment algorithm, reported in Figure 1, takes as input a job T, a cluster S and a lower bound l for the expected job latency that will be employed to drive the assignments computed as output. The initial value of the job latency limit l is equal to the lower bound l^{**}, computed as shown in the previous section. The main loop of the algorithm iterates until all tasks are assigned to a server and is structured in three phases each of which acts on a different partition of the set of slave servers. At the beginning, the following subsets of servers and tasks are considered. S_{inf} includes all servers that can execute at least one local task within the limit l but not a remote one, while S_{sup} includes those servers that can execute at least one remote task within the limit l (lines **3**–**4**). Servers in the complementary set $S_{\text{busy}} = S \setminus (S_{\text{inf}} \cup S_{\text{sup}})$ will not be considered until the limit l for the job latency is increased, thus leading to consider them in S_{inf} or S_{sup} in subsequent iterations of the main loop. The job T is partitioned in two subsets: T_{loc} and T_{rem}, where T_{loc} includes any task that can be executed on at least one server in $S_{\text{inf}} \cup S_{\text{sup}}$ and T_{rem} includes any task that can only be executed remotely before the limit l (lines **5**–**6**).

The body of the main loop is divided in three phases. In the first phase (lines **7**–**15**), we assign as many tasks as possible from T_{loc} to servers in S_{inf}, without exceeding the limit l. The tasks from T_{loc} are selected in ascending order of $|\rho(t)|$ (i.e., ranked by the number of servers where they can access data locally), so as to assign first those tasks that can only be executed on few servers, and are therefore more likely to cause violations of the target latency l. This is due to the fact that the initial value of l is l^{**}, which has been computed without taking into account the effect of many tasks having data on a small group of servers. In the second phase (lines **16**–**30**), we assign tasks from T_{rem} to servers in S_{sup}, without exceeding the limit l. During the first iteration of the main loop, all tasks from T_{rem} might be assigned, because the limit l is initially set to l^{**}, which guarantees that all tasks that need to be executed remotely can be completed within l^{**}. In the third phase (lines **31**–**41**), we assign as many tasks as possible from T_{loc} to servers in S_{sup}, without exceeding the limit l. Finally, if some tasks are still unassigned, the algorithm increases the limit l by one unit, recomputes the four subsets (T_{loc}, T_{rem}, S_{inf}, S_{sup}) and iterates the three phases. Note that the second phase forces the assignment of as many remote tasks as possible, employing time that could be usefully exploited by other jobs in return for a potentially very low latency gain. Thus, the algorithm triggers the execution of the second phase through a *threshold function* (ConsiderRemoteAssignments at line **16**) that is *true* until a given latency limit is reached, and *false* thereafter.

3.5 Example

To understand the behavior of the LABL algorithm, we compare it to the locality-aware round-robin [15] and flow-based algorithms [6], using a limited number of servers, $|S|$=10, and tasks, $|T|$=20. The task execution times are set at w_{loc}=1, w_{rem}=3. Figure 2 reports the considered data placement, with a maximum data replication factor of 2. Figure 3 reports assignments generated by

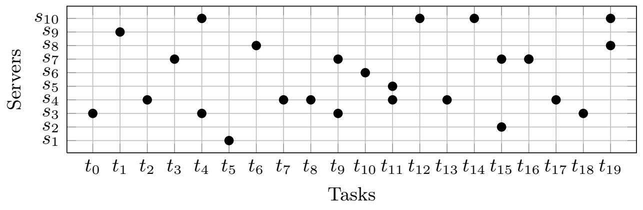
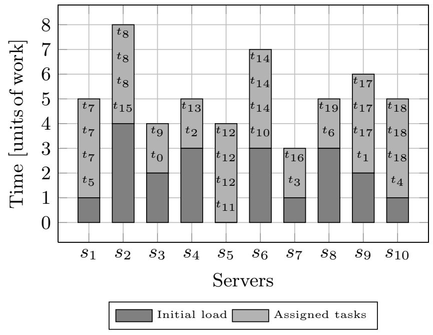


Fig. 2. Data Placement


(a) Round-robin: $C_{\texttt{rr}}(T)$=32 units of work

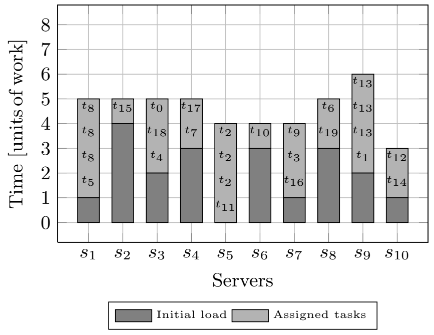
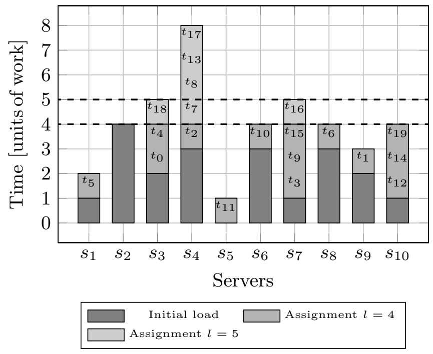

(b) Flow-based: $C_{\texttt{flow}}(T)$=26 units of work

Fig. 3. Round-robin (3a) and Flow-based (3b) Assignments


(a) $C_{\texttt{LABL}}(T)$=20 units of work. Remote assignment performed with load threshold l=4

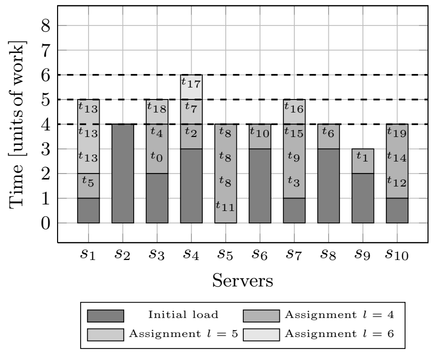

(b) $C_{\texttt{LABL}}(T)$=24 units of work. Remote assignments performed with load threshold $l \in \{4, 5\}$

Fig. 4. Locality Aware & Bounded Latency Task Assignments

the round-robin algorithm [15] and the flow-based algorithm [6], while Figure 4 shows assignments generated by the LABL algorithm, when the execution of the second phase is stopped after the first iteration. The round-robin algorithm cycles through the list of servers in a pre-determined arbitrary order until all tasks have been assigned (in the example, starting from s_1, then s_2, s_3, etc.). At each step, a task is assigned to a server. The algorithm tries to exploit the data placement by assigning a local task to the current server. If this is not possible, a remote task is assigned. The greedy choices of the round-robin algorithm results in a final assignment (see Figure 3a) with high job latency and high resource consumption ($l_{\texttt{rr}} = 8$, $C_{\texttt{rr}}(T) = 32$). The approach reported in [6] improves the round-robin strategy and describes an algorithm that allows to choose the minimum latency assignment among a list of $|T|$ possibilities. Each assignment is computed through a flow-based approach to maximize the assignment of local tasks (while limiting the load of the corresponding servers under a temporary threshold) followed by a greedy strategy necessary to complete the assignment of remote tasks. Figure 3b shows the assignment resulting from the aforementioned strategy ($l_{\texttt{flow}} = 6$). We note that the greedy choice, applied to assign the remote tasks, can often lead to resource consumption higher than the minimal one: $C_{\texttt{flow}}(T) = 26 > 20$.

Figure 4 depicts the assignments computed by the LABL algorithm when taking as input an initial job latency limit l=4. The algorithm exhibits different behaviors in terms of total job latency and minimization of resource allocation depending on the configuration of the *threshold function* (see Figure 1, line **16**: ConsiderRemoteAssignments) that stops the execution of the second phase of the algorithm from a specified iteration on. Figure 4a, shows the assignments obtained when the second phase is executed only at the first iteration.

Note that this has no effect on the final assignment since, at the first iteration, there is no tasks that needs to access data remotely. Indeed, the initial servers load specified in Figure 4a suggests that only tasks local to server s_2 may be considered for remote assignment. The data placement function specifies that t_{15} is the only task that can be assigned on s_2, however t_{15} is also local to server s_7. Thus, t_{15} has to be assigned on s_7. The final assignment in Figure 4a uses resources sparingly ($C_{\texttt{LABL}}(T) = 20$, equal to the minimum), at the cost of an increased latency ($l_{\texttt{LABL}} = 8$). To decrease latencies, it is necessary to consider the explicit handling of remote tasks up to the second iteration (Figure 4b). This allows to assign tasks t_8 and t_{13} remotely, contributing to lower the overall latency, at the cost of an increased resource usage. With respect to the assignment found by the flow-based algorithm, we achieve the best possible combination of job latency $l_{\texttt{LABL}} = 6$ and resource usage $C_{\texttt{LABL}}(T) = 24$.

3.6 Formal Properties of LABL Task Assignment Algorithm

In this section, we analyze the properties the LABL Task Assignment algorithm. We first prove that the algorithm can be configured by manipulating the ConsiderRemoteAssignments threshold function to achieve strong properties on load balance and resource usage. Subsequently, we analyze the computational complexity of the LABL algorithm.

Theorem 1. *Under the condition that* ConsiderRemoteAssignments *is* true *for all iterations of the main loop, the LABL Task Assignment algorithm produces an assignment $\mathcal{A}_{LABL}$ with*

$$\max_{s\in S}\phi(s) \leq \min_{s\in S}\phi(s) + w_{\text{rem}}$$

Proof. Let $s_{\max}$ be one of the servers such that the latency of the computed assignment is $l_{\texttt{LALB}} = \phi(s_{\max})$ and $s_{\min}$ be another server such that the execution of the tasks on it makes its final completion time $\phi(s_{\min})$ equal to the minimum latency among the servers in S. The proof will be developed through a *reductio ad absurdum*. Assume that $\phi(s_{\max}) > \phi(s_{\min}) + w_{\text{rem}}$ holds at the end of the LABL algorithm execution, and that the latency of the computed assignment is $l_{\texttt{LABL}} = l_{\texttt{out}}$. Such an hypothesis implies that in the last-but-one iteration of the outer loop of the LABL algorithm, there was a number n of tasks that could not be assigned within the latency limit $l = l_{\texttt{out}} - 1$. In the case $n = 1$, this task would have been assigned to the server $s_{\min}$ in the phase II of the algorithm, as the hypothesis guarantees enough resources for the remote execution of it. This contradicts the initial assumption as the aforementioned last iteration would not have occurred, and therefore the latency of the computed assignment would have been $l_{\texttt{LABL}} = l_{\texttt{out}} - 1$.

In case $n > 1$, each task can be sequentially assigned for remote execution to a server, starting from the one having workload equal to $\phi(s_{\min})$, as long as the number of tasks and the number of servers satisfying the condition $\phi(s) + w_{\text{rem}} \leq l_{\texttt{out}} - 1$ allows the assignments. If all tasks are assigned, then the last iteration would not have occurred, thus having the same conditions of the former case. Otherwise, the remaining tasks must be assigned at the next iteration when $l = l_{\texttt{out}}$ as the servers in the last-but-one iteration could have included only tasks requesting an execution time in $[w_{\text{loc}}, w_{\text{rem}} - 1]$ which is not obviously the case. In the last iteration there would have been only servers that could satisfy assignments of tasks with an execution time ranging from w_{loc} to w_{rem}. Therefore the difference between the maximum and the minimum workload would be $\phi(s_{\max}) - \phi(s_{\min}) \leq w_{\text{rem}}$, that contradicts the hypothesis.

Corollary 1. *If Theorem 1 holds and the server $s_{\min} \in S$ with minimum workload satisfies the condition $\phi(s_{\min}) \leq l^{**}$, then the optimal latency l^{opt} for the given assignment problem is bounded as: $l_{\texttt{LABL}} - w_{\text{rem}} \leq l^{\texttt{opt}} \leq l_{\texttt{LABL}}$.*

Proof. The lower bound given by l^{**} is lesser than or equal to $l^{\texttt{opt}}$ by definition, while $l^{\texttt{opt}}$ is, in turn, lesser than or equal to the latency limit computed by the LABL algorithm: $l^{**} \leq l^{\texttt{opt}} \leq l_{\texttt{LABL}}$. Now, if Theorem 1 holds, then $l_{LABL} = \phi(s_{\max})$ and $\phi(s_{\min}) \geq l_{\texttt{LABL}} - w_{\text{rem}}$. Therefore, noting that l^{**} must be greater than or equal to $\phi(s_{\min})$, leads to the thesis.

Theorem 2. *The LABL Task Assignment algorithm, under the condition that* ConsiderRemoteAssignments *is false for all values of $l > l^{**}$, produces an assignment $\mathcal{A}_{\texttt{LABL}}$ with a total resource usage*

$$C_{\texttt{LABL}}(T) \leq l^{**} \times |S| - \sum_{s\in S}\phi_s$$

Proof. If ConsiderRemoteAssignments is false for all l except l^{**}, the second phase of the LABL algorithm is executed only once, that is the assignment of remote tasks is performed only in the first iteration (i.e., when $l = l^{**}$).

If all the tasks are assigned in the first iteration (that is, the algorithm computes a final latency $l_{\mathtt{out}} = l^{**}$) then the resource allocation in terms of units of work is due to the servers in $S_{\text{sup}} \cup S_{\text{inf}} = S \backslash S_{\text{busy}}$, as in S_{busy} there are only servers with a workload that doesn't allow to cope with either local or remote tasks. Therefore the following relation holds:

$$\sum_{s \in S_{\text{sup}} \cup S_{\text{inf}}} (l^{**} - \phi_s) \geq \sum_{s \in S} (l^{**} - \phi_s)$$

The term in the right side of the previous inequality ($C_{\mathtt{LABL}} \leq \sum_{s \in S} (l^{**} - \phi_s) = l^{**} \times |S| - \sum_{s \in S} \phi_s$) is always smaller than the left one, as the workload of servers in S_{busy} is by definition greater than or equal l^{**}.

If the LABL assignment algorithm terminates with $l_{\mathtt{out}} > l^{**}$, then through remembering that the latency limit given by l^{**} guarantees (by definition) that the whole cluster S can allocate all the remote tasks (see the first condition in the definition of l^{**} in Section 3.3), and following the theorem hypothesis the assignment of tasks in the first and third phase of the algorithm will proceed through allocating the tasks locally, it is easy to infer that the whole number of units of work actually spent by the cluster ($C_{\mathtt{LABL}}$), at the end of the computation, will not be greater than $l^{**} \times |S| - \sum_{s \in S} \phi_s$.

Theorem 3. *The LABL Task Assignment algorithm operates in time*

$$\mathcal{O}\left(\log |T| \times |T| \times \max_{t \in T} |\rho(t)|\right)$$

where $|T|$ is the number of tasks and $\max_{t \in T}(|\rho(t)|)$ is the maximum number of data copies available for a task.

Proof. We represent $\rho(t)$ as adjacency lists sorted by server load and $\rho^{-1}(s)$ as adjacency lists sorted by $|\rho(t)|$. The sorting of subsets of T can be performed employing a counting sort algorithm, and has therefore $\mathcal{O}(|T| + \max_{t \in T} |\rho(t)|)$ complexity, since there are at most $\max_{t \in T}(|\rho(t)|)$ keys. The sorting of subsets of S can also be performed employing a counting sort algorithm, and has therefore $\mathcal{O}(|S| + \max_{s \in S} \phi(s))$ complexity, since there are at most $\max_{s \in S} \phi(s)$ keys. Note that the maximum values of $|\rho(t)|$ and $\phi(s)$ are two orders of magnitude smaller than $|T|$ and $|S|$ in real world cases, so using counting sort or other distribution sort algorithms is a reasonable choice. In particular, $\phi(s) \leq \max\{\phi_s, l^{**}\}$ initially, and $\phi(s) \leq \max\{\phi_s, l\}$ in successive iterations.

Computing the four sets S_{inf}, S_{sup}, T_{loc} and T_{rem} amounts to a single scan of S and T. Since in general $|S| < |T|$, the construction is overall $O(|T|)$. The first phase scans the entire S_{inf}. At most w_{rem} tasks are assigned for each $s \in S_{\text{inf}}$, since doing otherwise would lead to violating the latency bound. The complexity of this phase is therefore $\mathcal{O}(|S|)$. The second phase scans the entire T_{rem}, and assigns all tasks to the least loaded servers in a round robin way. The complexity of this phase is straightforward, as it performs $\mathcal{O}\left(|T_{\text{rem}}|\right)$ operations, which is also $\mathcal{O}\left(|T|\right)$. While the complexity of the third phase, as explained in Figure 1 is $\mathcal{O}\left(|T|\right)$, it is possible to implement it by iterating on the servers in S_{sup} and assigning as many task to each server as it can handle within the latency bound. This leads to a complexity of $\mathcal{O}\left(|S|\right)$.

Overall, we have a complexity that is bounded by $\mathcal{O}\left(|T| + \max_{t\in T}|\rho(t)|\right) + \mathcal{O}\left(|S| + \max_{s\in S}\phi(s)\right)$ for each iteration of the main loop. Since we increase l by one at each iteration, the number of iterations of the main loop is given by $l_{\texttt{LABL}} - l^{**}$, where $l_{\texttt{LABL}}$ is the latency of the assignment. Note that, even if we allocated every task remotely, $l_{\texttt{LABL}}$ would be limited by

$$l_{\texttt{LABL}} \leq \Big(w_{\text{rem}}|T| + \sum_{s\in S}\phi_s\Big)/|S|$$

Considering that $l^{**} \leq (w_{\text{loc}}|T| + \sum_{s\in S}\phi_s)/|S|$, it follows that $l_{\texttt{LABL}} - l^{**} \leq (w_{\text{rem}} - w_{\text{loc}})|T|/|S|$. In general, it can be assumed that $|T| \simeq c|S|$, where c is a small factor typically ranging in $\{2\ldots10\}$, therefore the outer loop is executed only a fixed number of times [4, 10]. However, we ensure this by means of the threshold limit of l imposed by ConsiderRemoteAssignments. Thereafter, we perform a reduced loop including only the first and third phases. This reduced loop, per se, has a complexity $\mathcal{O}\left(|T|^2\right)$, but it can be usefully restructured w.r.t. the general presentation to reduce the complexity. Specifically, since we are now only assigning tasks t to servers in $\rho(t)$, we can simply work as follows: for each $s \in S$, compute a set $R_s = \{t \in \rho^{-1}(s) \text{ if } t \in T\}$, and sort each set by $|\rho(t)|$.

We now iterate over the servers $s \in S$ in a round-robin way, removing one element of R_s at each iteration and assigning it to s if it has not been already assigned. This guarantees completion in:

$$\mathcal{O}\Big(\log|T| \times \sum_{s\in S}|R_s|\Big) = \mathcal{O}\Big(\log|T| \times |T| \times \max_{t\in T}|\rho(t)|\Big)$$

4 Simulation Results

We conducted an experimental campaign to compare the behavior of the LABL Task Assignment with the round-robin and flow-based algorithms. We employed as a starting point a real-world configuration from [4], which provides statistical data on the execution of MapReduce jobs at Google during an entire month.

The experiments are conducted in a simulation environment, scheduling one job on a set of servers having an existing workload. This is done to simulate the online scheduling process: given the mean inter-arrival time of 2-3 minutes reported in [3, 10], the job tracker will have completed the scheduling process of the job before a second one arrives. On the other hand, due to the long computation times, previously scheduled jobs will still be active while the new one is being scheduled. The simulation assumes tasks to require the same time w_{loc} to be executed on any server storing the necessary data. Since the time

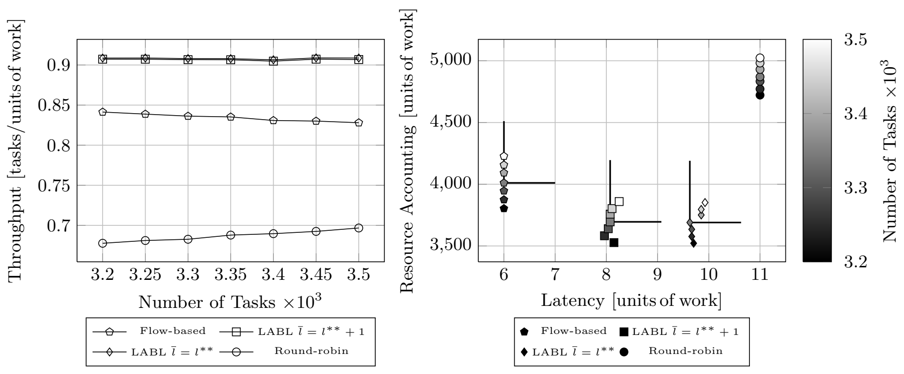


Fig. 5. Performance of Analyzed Algorithms

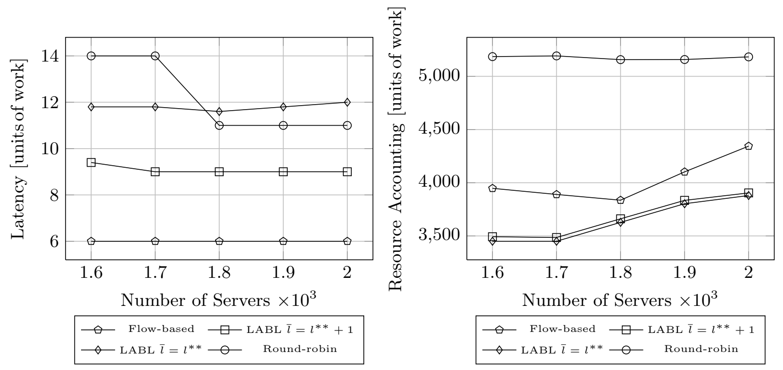
w_{loc} also represents a *unit of work*, we will consider w_{loc}=1 in all experiments. Whenever a task is assigned to a server that does not have the required data, the data must be fetched, leading the execution time to increase to w_{rem}. We set w_{rem}=3 in all experiments, following the same approach as [6]. We explore a configuration space considering a number of servers $|S|$=$\{1600, \ldots, 2000\}$ and a number of tasks $|T|$=$\{3200, \ldots, 3500\}$, though we will only show subsets of the overall configuration space in some experiments for the sake of clarity. The data placement is randomly determined such that $|\rho(t)|$ is in the range $[1, \rho_{\max}]$ for all tasks, where $\rho_{\max}$ is a parameter fixed at 4 in all experiments, except when evaluating the sensitivity of the algorithms to the replication factor. In all the experiments, the initial load is randomly assigned, within the range $[0, 5]$. In all cases, the reported data has been obtained as the average of the results gathered from 30 runs of the same experiment.

4.1 Performance Overview

The experiment reported in Figure 5 compares the effectiveness of the LABL Task Assignment with both the round-robin and flow-based algorithms, in terms of throughput, resource accounting and latency. We explore a configuration space with $|S|$=2000, $|T|$=$\{3200, \ldots, 3500\}$. Data for the LABL algorithm are reported for configurations with threshold latency $\bar{l}$ set to l^{**} and $l^{**} + 1$.

Figure 5a shows the throughput achieved by the three algorithms. The LABL algorithm, in both versions, yields a better throughput, i.e. the task assignment is able to consistently save resources, leaving more server time for other jobs.

Figure 5b reports in a scatter-plot the latency and resource consumption obtained by the three algorithms on the 2000 servers cluster, showing increasing number of tasks in the job by lighter shades. Figure 5b shows that the flow-based algorithm consistently obtains optimal latencies, while the LABL algorithm reduces resource usage. The LABL algorithm and the flow-based algorithm produce solutions that are Pareto-optimal, while the round-robin algorithm produces solutions that are Pareto-dominated by all the others.



(a) Latency (b) Resource accounting

Fig. 6. Scalability of the Analyzed Algorithms

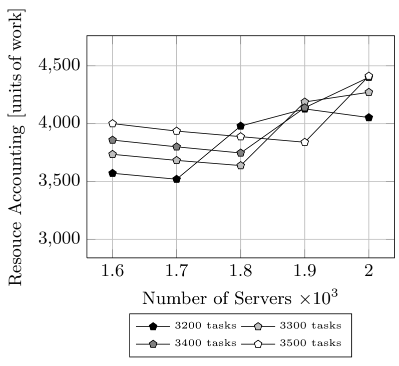
On the overall, the flow-based and LABL algorithms produce solutions of interest respectively to optimize latency and resource usage. However, the flow-based algorithm has a higher computational complexity, $\mathcal{O}(|T|^2 \times |S|)$ [6], making the LABL solution more attractive.

4.2 Scalability

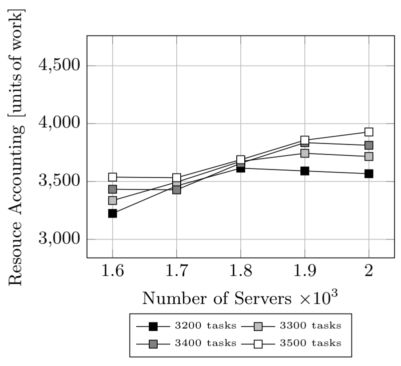
The experiment reported in Figure 6 evaluates the robustness of the four algorithms to changes in the availability of servers. Given a set of tasks T, $|T|$=3450, a data placement, and an initial workload, we progressively increase the number of servers that are available for scheduling from a minimum of $|S|$=1600 to a maximum $|S|$=2000. A desirable property for the scheduling algorithm is that the number of available servers has only limited impact on the latency — assuming there are enough servers to actually execute the job. Figure 6a shows that only the round-robin algorithm is significantly impacted by the change in server availability. This is because the round-robin algorithm makes greedy choices, which easily prove suboptimal. The other three algorithms behave in a more graceful way, as their greedy choices are less aggressive — all four algorithms have greedy components within their heuristics, to limit the complexity, but the greedy component is dominant only in the round-robin algorithm. The LABL algorithm produces Task Assignments with higher latencies than the flow-based algorithm. This is expected since, as shown in Section 4.1, the LABL algorithm trades off latency to save resources. Figure 6b shows the impact of server availability on the resource usage. The impact is minimal on the round-robin algorithm, while the other three algorithms all tend to consume more resources when these are available, by placing remote tasks on free servers in an attempt to reduce latency. However, the LABL algorithm, in both versions, always outperforms the flow-based algorithm, thanks to its greater focus on reducing resource usage.

4.3 Sensitivity Analysis

The experiments reported in Figure 7 and Figure **??** evaluate the sensitivity of resource usage to, on one hand, the number of tasks to execute and the number of



(a) Flow-based



(b) LABL $\bar{l} = l^{**}+1$

Fig. 7. Resource Awareness of Analyzed Algorithms

available servers, and, on the other hand, the replication factor, i.e. the average number of copies of the data accessed by a task.

In the first case, only the resource accounting for the flow-based (Figure 7a) and LABL algorithm with $\bar{l} = l^{**} + 1$ (Figure 7b) are shown, as these algorithms have proven to be the most effective ones (see Figure 6). Figure 7 depicts a family of curves representing resource accounting as a function of the number of servers ($|S|$=$\{1600, \ldots, 2000\}$), considering the number of tasks $|T|$=$\{3200, \ldots, 3500\}$ as a parameter. As expected, the LABL algorithm consumes less resources. The results also show that the behavior of the LABL algorithm is much more stable. Moreover, the flow-based algorithm is characterized by a higher resource usage when scheduling more tasks. Focusing on the replication factor, Figure **??** shows only the resource accounting employed by the flow-based and LABL algorithm (with $\bar{l} = l^{**} + 1$), as a function of the cluster size. The round-robin strategy is not considered since it consistently employs a higher number of resources (see Figure 6b). We vary the maximum replication factor $\rho_{\max}$ from 2 to 7, so that the average replication factor ranges in $[1.5, 4]$. Thus, the generated data placements have $|\rho(t)|$ uniformly distributed in the range $[1, \rho_{\max}]$ for all tasks. The results show that the LABL algorithm is less sensitive to the replication factor than the flow-based one. The flow-based algorithm takes greater advantage from the increased locality given by the presence of more replicas of each data item, but the LABL algorithm is still able to achieve a lower resource usage. Note that a higher replication factor does impact on the overall costs — keeping up to date copies of the data across the network is bound to have a significant communication cost, so the ability to achieve good resource utilization with a low replication factor is a strong asset of the LABL algorithm.

4.4 Discussion

We will now discuss the interactions of the LABL algorithm with other scheduling goals such as fairness and adaptivity, as well as potential optimizations.

Scheduling for Fairness The *fairness* property is often desirable in large-scale clusters that are accessed by multiple users. That is, the applications submitted by any user should not be delayed indefinitely. Online scheduling strategies, such as the LABL algorithm, can be integrated into higher level policies aimed at providing such fairness guarantees, that is, at user-application scheduling level rather than at task-scheduling. Indeed, the LABL algorithm could effectively replace the round-robin algorithm that is used as the task assignment component of the *Hadoop fair scheduler* [6,15].

Scheduling Jobs from Multiple Applications It is possible that, for a given job, some servers of the cluster have no copies of the required data for any of its tasks — or a set of servers $S' \subset S$ has only copies of data needed for a set of tasks $T' \subset T$, but $|T'| < |S'|$, leaving $|S'| - |T'|$ servers idle. In this case, the servers cannot be used to run a local task, either leading to execution delays, if they are used to run a remote task, or to an under-utilization of resources. To further improve resource utilization and throughput, it is possible to schedule jobs from multiple applications at the same time, as these are likely to use different data sets. It is worth noting, however, that scheduling multiple jobs increases the throughput at a cost in latency. The LABL scheduling algorithm, however, can easily handle the schedule of sets of tasks belonging to different jobs coming from independent applications, through simply merging the two sets. The key issue is selecting jobs that map on data held in different servers, so as to allow servers that cannot run tasks locally for one job to be used for another job.

Adaptive Scheduling A latency-aware scheduling is more attractive when the cluster is under-utilized, as it allows to minimize application latency, providing a better response time to the user. On the other hand, a resource-aware scheduling becomes increasingly important as the cluster utilization grows. Indeed, in a cluster under a heavy workload, a scheduling policy that favors latency may easily lead to low availability for other jobs. A common solution is to artificially limit the amount of resources that a single job can take. The LABL algorithm does that, by construction, optimizing the resource accounting of the scheduled job, while still providing a strong latency limit. Thus, it adapts better to workload variations, as shown in Section 4.3.

5 Related Work

The MapReduce programming model has been formalized in a number of ways. In [9] MapReduce computations have been compared to the PRAM model, focusing on analyzing how PRAM algorithms can be expressed using MapReduce. Among the studies on task assignment, in [11] the authors focus on allocating tasks of multiple jobs in both on-line and off-line scenarios, providing a generalization of the Flexible-Flow Shop problem. However, the authors do not take into account the impact of data placement, which is critical due to the size of the exchanged data. In [6] the Hadoop round-robin based task allocator is compared with a flow-based task allocator, showing that careful consideration of data placement allows to limit job latency. An in-depth comparison with both algorithms is provided in Section 4. Job latency reduction has been tackled in [19] considering a production-quality scenario, showing how careful job speculation helps on limiting the latency penalty introduced by straggled tasks (i.e., remotely executed tasks on the critical path), at the cost of an increased resource consumption. This technique, while applicable to all tasks, is more effective on *reduce* tasks, since map tasks are much less likely to be straggled. In a typical MapReduce implementation, the set of available resources is equally exposed to all jobs. In [14], on the other hand, a different processing resources are exposed to each job depending on its workload profile in terms of CPU, disk and memory usage. Thus, a task tracker can maximize the use of its resources through executing tasks from jobs with different profiles. This scheme can be easily combined with our own, since in our approach the set of resources is an input parameter, whilst the key aspect of [14] is the definition of the resource set for each job profile. In [16], a framework to estimate the latency of a MapReduce job as a function of the employed resources is introduced. The scheme is based on a job profile obtained through the execution of the same job on a smaller data set. This work, while not directly related to our own, could be adapted to provide stronger latency bounds for task assignment. This solution, however, would incur in the cost of job profiling. In [17], FLEX, a scheduler for MapReduce systems, is proposed as a replacement for the Hadoop fair scheduling algorithm. With respect to our work, FLEX does not take into account data locality, and works on multiple jobs at the same time in an epoch-based scheme. Similarly, in [18] multiple jobs are managed, aiming at fairness and data locality, but with no latency guarantees. The task assignment problem is common to all Data-Intensive Scalable Computing schemes. However, the solutions need to be tailored to the specific setup: e.g., [12, 13] deal with *cloud*-based MapReduce services, which rely on a heavy use of virtualization techniques. Virtualization is not attractive for every Data-Intensive Scalable Computing scenario, due to the need to spawn new virtual machines at high frequency — job completion times follow a long tailed distribution, with 80% of the successful jobs completing within 6 minutes, as shown in [10] for a 10-month timeframe on a production Yahoo! Hadoop cluster. In [8], on the other hand, a typical cluster of commodity machines is used to run tasks with dependencies, leading to different problems, such as the need to keep dependent tasks on near machines to minimize communications.

6 Concluding Remarks

We presented an algorithm for assigning the tasks of a job to servers in a MapReduce cluster. The proposed algorithm balances the tradeoff between latency and resource consumption. Simulation results support the insight that a practical implementation would benefit from the proposed approach. Future works include integrating the LABL task assignment algorithm within a higher level job scheduling framework, which would also manage fault tolerance issues. In addition, as a further refinement of the proposed technique, the cluster interconnect topology will be taken into account to model the remote execution time.

References

1. Bortnikov, E.: Open-source Grid Technologies for Web-scale Computing. SIGACT News 40(2), 87–93 (2009)
2. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: Easy, Efficient Data-parallel Pipelines. In: PLDI. pp. 363–375 (2010)
3. Chen, Y., Ganapathi, A., Griffith, R., Katz, R.H.: The Case for Evaluating MapReduce Performance Using Workload Suites. In: MASCOTS. pp. 390–399 (2011)
4. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI. pp. 137–150 (2004)
5. Dean, J., Ghemawat, S.: MapReduce: a Flexible Data Processing Tool. Commun. ACM 53(1), 72–77 (2010)
6. Fischer, M.J., Su, X., Yin, Y.: Assigning Tasks for Efficiency in Hadoop: Extended Abstract. In: SPAA. pp. 30–39 (2010)
7. Hey, A.J.G., Trefethen, A.: The Data Deluge: An e-Science Perspective. In: Berman, F., Fox, G.C., Hey, A.J.G. (eds.) Grid Computing - Making the Global Infrastructure a Reality, pp. 809–824. John Wiley & Sons, Inc., New York, NY, USA (2003)
8. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: Fair Scheduling for Distributed Computing Clusters. In: Matthews, J.N., Anderson, T.E. (eds.) SOSP. pp. 261–276. ACM (2009)
9. Karloff, H.J., Suri, S., Vassilvitskii, S.: A Model of Computation for MapReduce. In: SODA. pp. 938–948 (2010)
10. Kavulya, S., Tan, J., Gandhi, R., Narasimhan, P.: An Analysis of Traces from a Production MapReduce Cluster. In: CCGRID. pp. 94–103. IEEE (2010)
11. Moseley, B., Dasgupta, A., Kumar, R., Sarlós, T.: On Scheduling in Map-Reduce and Flow-Shops. In: Rajaraman, R., Meyer auf der Heide, F. (eds.) SPAA. pp. 289–298. ACM (2011)
12. Palanisamy, B., Singh, A., Liu, L., Jain, B.: Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud. In: Lathrop, S., Costa, J., Kramer, W. (eds.) SC. p. 58. ACM (2011)
13. Park, J., Lee, D., Kim, B., Huh, J., Maeng, S.: Locality-aware Dynamic VM Reconfiguration on MapReduce Clouds. In: HPDC. pp. 27–36 (2012)
14. Polo, J., Castillo, C., Carrera, D., Becerra, Y., Whalley, I., Steinder, M., Torres, J., Ayguadé, E.: Resource-Aware Adaptive Scheduling for MapReduce Clusters. In: Middleware. pp. 187–207 (2011)
15. The Apache Software Foundation: Hadoop MapReduce. `http://hadoop.apache.org/mapreduce` (September 2012 (retrieved))
16. Verma, A., Cherkasova, L., Campbell, R.H.: Resource Provisioning Framework for MapReduce Jobs with Performance Goals. In: Middleware. pp. 165–186 (2011)
17. Wolf, J.L., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.L., Balmin, A.: FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In: Middleware. pp. 1–20 (2010)
18. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In: EuroSys. pp. 265–278 (2010)
19. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R.H., Stoica, I.: Improving MapReduce Performance in Heterogeneous Environments. In: Draves, R., van Renesse, R. (eds.) OSDI. pp. 29–42. USENIX Association (2008)