Spark Task Scheduling Strategy Based on Multidimensional Load Sensing

Congyang Wang, *Member, IAENG,* Qingsong Xu, Haifeng Fei, Han Li*, Yanhao Zhang and Junyang Yu

***Abstract*—In the Spark parallel computing framework, the default task scheduling algorithm primarily follows the principle of data locality. It pursues a higher data localization level through a delay scheduling strategy but ignores differences in node computing capacities and real-time load states, leading to low task execution efficiency. To address this issue, this paper proposes a multi-dimensional load-aware Spark task scheduling strategy (MDLS). The strategy first evaluates node computing capacities and monitors real-time loads. Then, it constructs a multi-dimensional load-aware task execution time prediction model by correlating and analyzing task characteristics and node states. Next, the Executor-Task scheduling problem is transformed into a Minimum Weighted Bipartite Graph Matching problem based on execution time. For Spark's batch execution characteristics, MDLS integrates the Longest Processing Time scheduling concept. Within each scheduling batch, the task scheduling problem is modeled as a fully weighted bipartite graph matching problem. An optimization model is constructed with the objective of minimizing task completion time, and the Kuhn–Munkres Algorithm is adopted to solve for the optimal scheduling scheme. Finally, the strategy is implemented in Spark-2.4.6 and evaluated via benchmark tests. Experimental results show that the MDLS strategy effectively alleviates cluster resource contention and significantly improves task execution efficiency: a 37.6% improvement over Spark's default mechanism and an 11.5% improvement over the RUPAM strategy.**

***Index Terms*—Spark; Task Scheduling; Minimum Weighted Bipartite Graph; Kuhn-Munkres; Dynamic Load.**

I. Introduction

WITH the proliferation of big data applications [1–5], MapReduce [6] has become the mainstream framework for large-scale data processing in industry [7], [8]. As an extension of the MapReduce paradigm, Apache Spark [9] is widely adopted in industry due to its in-memory computing capabilities, which significantly enhance data processing efficiency by leveraging memory resources effectively [10].

As illustrated in Fig. 1, in a Spark application,Resilient Distributed Dataset(RDD) construct Directed Acyclic Graphs(DAGs) through a series of operations such as join, groupBy, and filter. The DAGScheduler component then partitions the DAG into distinct stages based on Shuffle operation characteristics, submitting partitioned tasks to the TaskScheduler as TaskSets. The TaskScheduler distributes these tasks to the eCluster manager, and Worker Nodes receive tasks to execute in thread pools. Upon job completion, Spark releases all resources occupied by the application.

In task scheduling, Spark employs a data locality-first delay scheduling strategy, prioritizing task execution on nodes with high data locality. However, this default strategy does not account for node computing capacity heterogeneity, real-time load dynamics, or task execution states within Executors. Additionally, data volume variations across tasks can cause load imbalance. The default task execution order does not adapt to task differences, resulting in suboptimal utilization of cluster resources and prolonged task completion time.

Spark tasks are divided into Map Tasks and Reduce Tasks, with shuffle operations as the boundary (Figure 2). In the Map phase, Map Tasks read and process data blocks to generate intermediate results. In the Reduce phase, Reduce Tasks retrieve all Map Task output data via shuffle operations. This many-to-many communication model creates a performance bottleneck in data transfer during the Reduce phase. Cross-node or cross-rack data transfer delays are significantly higher than local data reading efficiency. Additionally, heterogeneity in node computing capacities (CPU, memory, and I/O performance)—combined with real-time load variations—causes dynamic fluctuations in node processing capabilities.

Spark's default task scheduling algorithm assumes node homogeneity, failing to account for cluster heterogeneity or dynamic changes in node resource utilization and real-time load. This design leads to low resource efficiency in heterogeneous environments, making it difficult to achieve load balancing and satisfy performance requirements for diverse computing scenarios. Although prior studies have proposed effective task scheduling algorithms [11], [12] and data locality-driven strategies [13], [14], they predominantly rely on greedy-based scheduling with static task/node characteristics. These approaches prioritize local optimization over global optimization, hindering synergistic optimization of data locality and resource allocation. Study [15] explores globally optimal task-locality allocation from a data locality perspective but ignores the impact of hardware and load heterogeneity on scheduling.

To address the limitations of existing research, this paper proposes a multi-dimensional load-aware task scheduling strategy (MDLS) that comprehensively considers cluster performance and load conditions. The main contributions are as

Manuscript received Apr 21, 2025; revised Aug 29, 2025. This work is supported by the National Natural Science Foundation of China (No. 92367302 and No. 92467103), the subproject "New Industrial Internet Service Security System" under the NSFC Major Research Program (No. 92367302), and the Henan Provincial Science and Technology Key Project (No. 242102210046).

Congyang Wang is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: wangcongyang@henu.edu.cn).

Qingsong Xu is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: 2670993885@qq.com).

Haifeng Fei is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: fhf@henu.edu.cn).

Han Li is an associate professor of School of Software, Henan University, Kaifeng 475004, China (corresponding author to provide phone: +86-13837853817; e-mail: lihan@henu.edu.cn).

Yanhao Zhang is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: zhangyanhao@henu.edu.cn).

Junyang Yu is a professor of the School of Software, Henan University, Kaifeng 475004, China (e-mail: jyyu@henu.edu.cn).

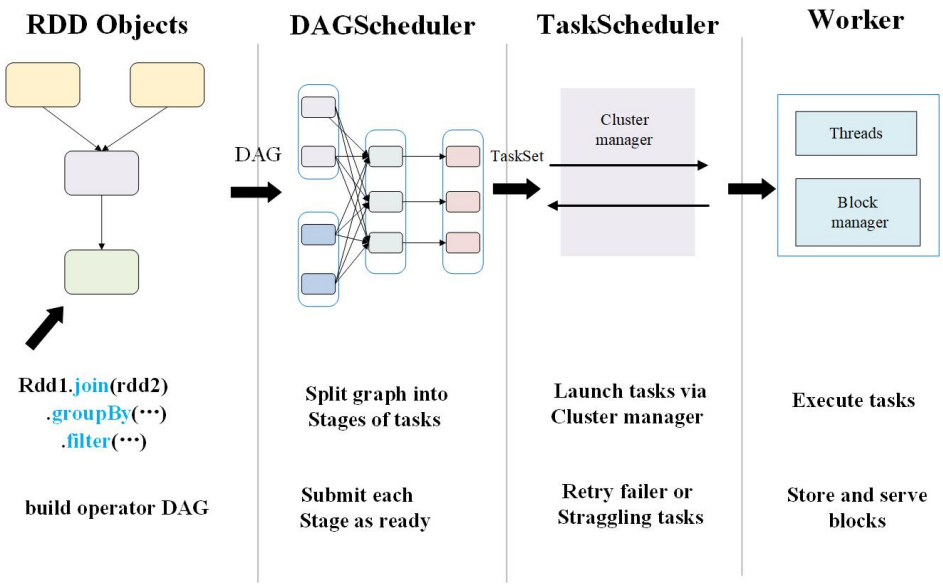

Fig. 1. Spark Task Transformation and Execution Process.

follows:

1. Task Execution Time Prediction Model: By synthesizing task characteristics, data distribution, node heterogeneity, and real-time load states, an accurate task execution time prediction model is built. This model quantitatively forecasts task execution times on different Executors, providing a reliable basis for scheduling decisions.

2. Multi-Dimensional Load-Aware Scheduling Strategy: The task scheduling problem is modeled as a Minimum-Weighted Bipartite Graph Matching problem. First, a scheduling matrix integrating node computing capacities and real-time load information is constructed. Then, a mathematical model is established to minimize task set completion time, enabling intelligent task-resource matching.

3. LPT-Based Complete Weighted Bipartite Graph Optimization: By combining the Longest Processing Time (LPT) strategy with Spark's batch scheduling mechanism, the weighted bipartite graph is extended to a Fully Weighted Complete Bipartite Graph structure. The Kuhn–Munkres (KM) algorithm is used to solve for the optimal scheduling scheme, optimizing task execution order while dynamically balancing the loads of cluster nodes.

4. Experimental Validation and Performance Analysis: To verify the effectiveness of MDLS, the strategy is evaluated through benchmark tests across multiple dimensions, including Map Stage completion time, Reduce Stage completion time, network bandwidth usage, and CPU utilization. Experimental results fully validate the strategy's feasibility and effectiveness in improving task execution efficiency and optimizing resource allocation.

The remainder of this paper is organized as follows: Section 2 reviews related research on Spark task scheduling; Section 3 details the model construction and algorithm design of the MDLS strategy; Section 4 analyzes experimental results and compares performance across different policies; and Section 5 concludes the paper.

II. Related Work

Task scheduling aims to assign a set of dependent or independent tasks to execute on cluster nodes with available resources. An effective task scheduling strategy must consider both the cluster resource load and the task execution time to find the optimal scheduling scheme, which is critical to improve overall job execution efficiency [16].

In the Spark platform, the default task scheduling algorithm prioritizes data location and uses a delay scheduling strategy to enhance data location (Figure 3). However, this algorithm has notable limitations, failing to account for node computing capacity heterogeneity or real-time load conditions [17]. It assumes homogeneous clusters with identical configurations and balanced loads, but in practice, heterogeneous clusters with computing capacity variations are common due to hardware upgrades, cluster scaling, and multi-role node deployments [16]. Spark's delay scheduling and round-robin scheduling are inefficient in heterogeneous environments: high-performance nodes idle after quick task completion, while low-performance nodes endure prolonged high loads, leading to unbalanced resource utilization and degraded cluster performance.

Non-local data exchange during Map and Reduce task execution is a common cause of performance degradation [18]. Zaharia et al. [19] proposed the Delay Scheduling Algorithm (DSA) for fair scheduling, which reduces network data transmission by balancing data transfer time and waiting time. When task node locality cannot be satisfied, DSA introduces appropriate delays to improve data locality. Guo et al. [20] transformed the scheduling problem into a Linear Sum Assignment Problem (LSAP) and proposed a node-locality-maximizing task scheduling algorithm. Tang et al. [23] modeled task scheduling as a graph problem, formulating optimal schemes by calculating task communication cost matrices to minimize communication overhead and enhance job performance.

However, a common limitation of these approaches is their neglect of node computing capacities and real-time load dynamics. Jin et al. [17] proposed BAR (Balance-Reduce), a

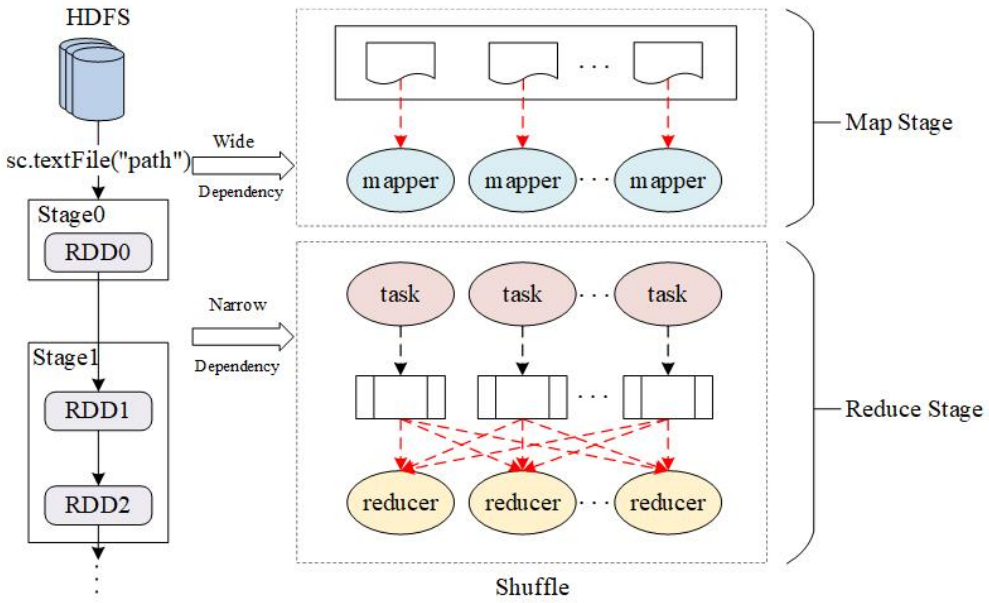

Fig. 2. Different Stages of Tasks and the Shuffle Process.

data locality-driven dynamic task scheduling algorithm that adjusts to node network conditions and workloads but ignores node computing capabilities. To address this, Naik et al. [11] developed a data locality-aware scheduling strategy based on node performance and load profiles, balancing locality and load by assigning data blocks to higher-capability nodes. Lu et al. [21] proposed a scheduling strategy weighing task complexity and node performance, assigning tasks to high-performance nodes based on heterogeneous node differences and resource demands to alleviate the ”barrel effect” in heterogeneous environments. With growing data volumes and cluster hardware aging, cluster heterogeneity intensifies; node capability-based task allocation optimizes resource utilization, balances loads, and reduces job runtime. Xu et al. [16] proposed the Resource and Usage Pattern-Aware Scheduling Management (RUPAM) strategy for heterogeneous environments. RUPAM integrates task resource and hardware characteristics, using a heuristic greedy algorithm to match tasks with nodes based on key factors (e.g., CPU, memory) via task sorting and priority queue matching, while preserving data locality. Although RUPAM improves program performance, its real-time dynamic adjustment capability is limited. Tang et al. [22] developed a dynamic memory-aware task scheduler that integrates CPU, memory, and other resources, adjusting task concurrency via real-time system feedback to optimize resource utilization and shorten application execution time.

Historical task execution information provides valuable a priori knowledge for dynamic scheduling. Li et al. [23] proposed an energy-aware task scheduling strategy using a historical policy table (recording execution times and energy consumption of past tasks) to assign tasks to optimal Executors, reducing job execution time and energy use. Tang et al. [24] developed a network-load-aware scheduling strategy based on historical execution data, categorizing tasks into network-intensive and network-idle states. This allows scheduling additional network-intensive tasks during network-idle phases and executing co-located CPU tasks as coarse-grained pipelined tasks to improve resource utilization. Zhang et al. [25] presented a delayed parallel Stage scheduling strategy using historical task scheduling data, optimizing CPU-network resource coordination to enhance task efficiency.

Task deadlines are another critical scheduling constraint. Unlike prior work focusing on static resource allocation, these studies introduce time-sensitive constraints but still lack dynamic adaptation to multi-dimensional load variations. Wang et al. [26] proposed a dynamic multi-task deadline scheduling algorithm, first determining task order via an improved priority strategy in the allocation phase, then allocating resources based on user time demands to reduce job runtime and meet deadlines. Wang [27] and Singh [28] incorporated ant colony algorithms to balance optimal completion times and node workloads. Gu et al. [29] developed a scheduling strategy addressing both data skewness and deadlines to minimize public cloud rental costs. Neciu et al. [30] proposed a real-time Earliest Deadline First (EDF)-based scheduling strategy for Spark jobs, dynamically adapting to heterogeneous cluster loads by managing job deadlines to reduce latency. Zhao et al. [31] optimized DAG-aware scheduling by integrating task dependencies with cache replacement policies, addressing inefficiencies in Spark’s cache management during DAG execution.

III. Spark Task Scheduling Strategy Based on Multidimensional Load Sensing

A. Task Execution Time Model

Spark task execution time is influenced by multiple factors: cross-node data shuffling occurs when task location mismatches data storage, with migration time determined by data volume, network bandwidth, and real-time load; node computing capacities, disk performance, and real-time load directly impact task execution duration [32]. Therefore, quantifying node computing capacities is essential to accurately predict task completion time.

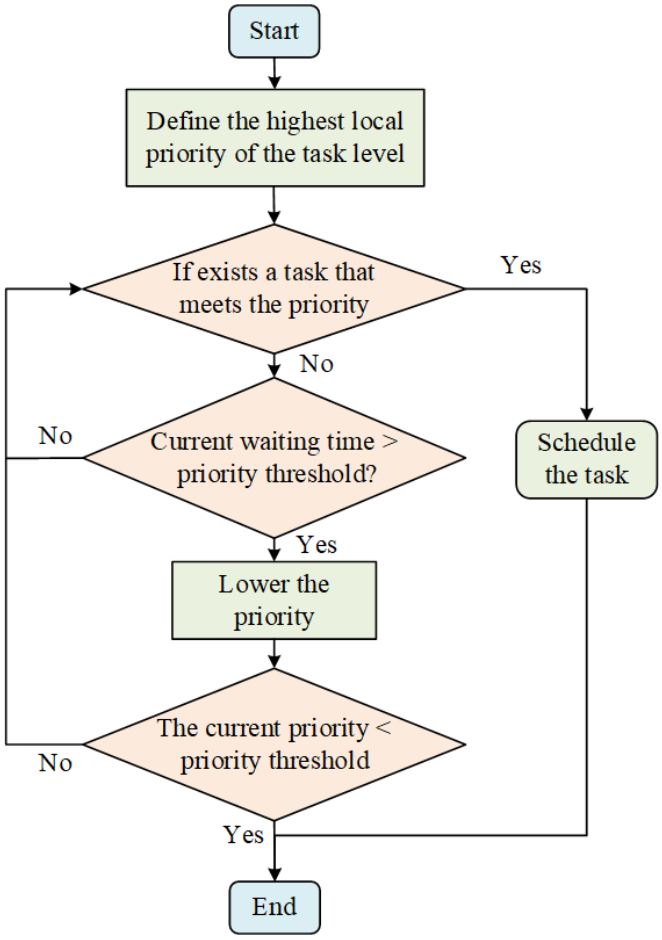

Fig. 3. Flowchart of Spark's Scheduling Strategy.

1) Node Performance Definition

During Spark execution, network data transfer is triggered when an Executor and task-execution data reside on different nodes. To quantify a node's ability to fetch remote data, this paper measures the average rate at which Executor e_j retrieves data from intermediate data nodes via repeated execution of a standard benchmark workload. The average rate at which remote data is retrieved by e_j to the local node for $task_i$ is defined as:

$$Net_{ij} = \frac{\sum_{k=0}^{num-1} net_{ijk}}{num}, j = 0, 1, ..., m \quad (1)$$

Where net_{ijk} denotes the network transmission rate between node j and the node k hosting the intermediate data fragment for $task_i$.

Computational capability $CalSpeed_j$ represents the current processing speed of node j. This is measured by executing multiple standard benchmark jobs with fixed data volumes and calculating their average execution speed:

$$CalSpeed_j = \frac{\sum_{l=0}^{num-1} Speed_l}{num}, j = 0, 1, ..., m \quad (2)$$

Multiple Executors on the same node share hardware resources, hence their execution rates are consistent.

Disk input/output (I/O) overhead is also a critical factor in Spark computations. Using the Linux dd command, we define disk write speed C_{write} and read speed C_{read} for node j as follows:

$$C_{write} = \frac{\sum_{m=0}^{num-1} C_w}{num}, C_{read} = \frac{\sum_{m=0}^{num-1} C_r}{num} \quad (3)$$

2) Dependency-Based Data Volume Sensing Algorithm

After Spark application jobs are transformed into Directed Acyclic Graphs (DAGs), they are divided into stages based on dependency levels. Specifically, narrow dependencies (e.g., map, filter) form continuous stages, while wide dependencies (shuffle operations) trigger stage boundaries. Analogous to the MapReduce framework, these include Map and Reduce stages. In the Map stage, each Map task reads a data partition and forms a "one-to-one" narrow dependency with the partition. In the Reduce stage, each Reduce task extracts partial intermediate data from Map task outputs, establishing a "one-to-many" wide dependency (shuffle dependency) with data partitions. While the Map stage's input data distribution is static (pre-known before task execution), the Reduce stage's data distribution is highly uncertain due to dynamically generated intermediate results, necessitating dynamic sensing algorithms for real-time computation. This subsection focuses on dependency-based data distribution sensing for Reduce-stage tasks.

In the Reduce stage, task-processed data originates from Map-stage task outputs, categorized as local (same node) or remote (cross-node) based on the Executor's location. Spark identifies Map-Reduce dependencies via shuffleId: Map task output locations and corresponding shuffleIds are stored in the Driver's MapOutputTracker. Before Reduce tasks start, they retrieve data locations from the Driver and fetch data from target nodes via BlockManagerClient. As Reduce tasks are distributed across nodes and Map tasks process varying data volumes, data pull volumes for different Reduce tasks fluctuate significantly.

Before submitting the Reduce-phase TaskSet, the amount of partitioned data and remote data pull per task can be accurately calculated using ShuffleId and Spark's RPC framework. The detailed implementation is outlined in Algorithm 1.

Through the above process, the data volume for Map and Reduce tasks is unified as $Size_{total}$, and the local data size for Reduce task is defined as $Size_{local}$. These serve as basic parameters for data distribution in subsequent execution time modeling.

3) Map Task Execution Time Model

The Map stage task execution time comprises three substages: data reading, data computation, and result output. For HDFS files, the number of Tasks by default equals the number of HDFS partitions. By querying HDFS partition locations via HDFS API , the data volume per Task is obtained as modeling input.

HDFS storage mechanisms directly impact data reading efficiency. For HDFS files, data read time depends on replica location and storage media (e.g., memory/disk). Based on how replicas are stored on the node where the Executor is deployed, the time to read data from the replica (memory/disk) to the Executor instance is evaluated as:

Algorithm 1 Dependency-Based Data Volume Sensing Algorithm

Input: Reduce task corresponding to stage ShuffleId: sf_{id}
Output: Partition data information for the task: $T_{P_{info}}$; The amount of partition data corresponding to the Reduce task: D_i;
Initialize: $T_{p\_info} \leftarrow \{\}$; $D_t \leftarrow \{\}$
$taskSet = getReduceTasks(sf_{id})$
for $task$ in $taskSet$ **do**
  $remoteData = 0$
  $taskPs = partitions.get\,(task, partitionNum)$
  for $partition$ in $taskPs$ **do**
    $T_{p\_info}[task.id].append\,(partition)$
    for $partition$ in $T_{p\_info}[task.id]$ **do**
      if $partition.Node \neq task.Node$ **then**
        $remoteData+ = mapSize$
        $D_t[task.id].remoteData = remoteData$
      end if
      $D_t[task.id].totalData+ = mapSize$
    end for
  end for
end for
return (T_{p_info}, D_t)

$$TimeLocal_{i,j} = \frac{Size_{total}}{S_{read} \times (1-\theta)} \tag{4}$$

Where S_{read} represents the memory or disk read speed, and θ denotes the current resource utilization rate. For local file systems (e.g., ext4), Spark requires files to be accessible via the same path across all nodes. In this case, data read time depends on the task's file size and the Executor's disk speed, i.e., S_{read} takes the value of C_{read}.

If task data replicas on other file systems are not present on node j, the data must be retrieved remotely over the network, and the retrieval time can be expressed as:

$$TimeRemote_{i,j} = \frac{Size_{total}}{net_{ijk} \times (1-\alpha)} \tag{5}$$

Where α represents the current network bandwidth utilization.

Once the data required by $task_i$ is loaded into the Executor's memory, the computation phase begins. The task computation time is calculated as:

$$TimeCal_{i,j} = \frac{Size_{total}}{CalSpeed_j \times (1-\beta)} \tag{6}$$

Where β denotes the current CPU utilization rate.

After Map task computation, intermediate data is written to disk for subsequent Reduce stage processing. Assuming a data scaling factor of γ and θ_{write} as the disk write bandwidth utilization ratio, the write time is:

$$TimeWrite_{i,j} = \frac{Size_{total} \times \gamma}{\mathrm{C}_{write} \times (1-\theta_{write})} \tag{7}$$

Therefore, the total execution time of Map $task_i$ on Executor e_j is:

$$\begin{aligned} Time_{i,j} = &[TimeLocal_{i,j}, TimeRemote_{i,j}] \\ &+ TimeCal_{i,j} + TimeWrite_{i,j} \end{aligned} \tag{8}$$

4) Reduce Task Execution Time Model

In the Reduce stage, each task fetches partitioned data from all Map stage output data. To avoid network congestion from many-to-many communication, Spark limits the data pulled per communication by the configuration parameter $MaxSizeInFlight$ (48 MB by default). Thus, the time for Reduce task to fetch data from remote node is:

$$\begin{aligned} TimeTrans_{i,j} &= \frac{Size_{\text{total}} - Size_{\text{local}}}{MaxSizeInFlight} \times \frac{MaxSizeInFlight}{Net_{ij} \times (1-\alpha)} \\ &= \frac{Size_{\text{total}} - Size_{\text{local}}}{Net_{ij} \times (1-\alpha)} \end{aligned} \tag{9}$$

Where $Size_{total}$ is the total data volume for $task_i$, $Size_{local}$ is the local data volume, and Net_{ij} is the network bandwidth utilization.

Remote-fetched data is stored in the Executor's memory buffer. When data exceeds the memory threshold, it spills to disk, incurring spill overhead. During spilling, Spark writes data to disk files in 10,000-entry batches. Assuming an average spill block size of $size_{spill}$ (per 10,000 entries), the number of spill operations is:

$$SpillNum = \frac{Size_{total} - Size_{local} - SpillThreshold}{Size_{spill}} \tag{10}$$

During memory data spill to disk, the disk write time is calculated as:

$$TimeSpill_{i,j} = \frac{SpillNum \times Size_{spill}}{C_{write} \times (1-\theta_{write})} \tag{11}$$

Upon data fetch completion, the task reads spilled data from disk and processes local data, both requiring disk access. The disk read time is:

$$\begin{aligned} TimeRead_{i,j} = &\frac{Size_{\text{total}} - SpillThreshold}{C_{\text{read}} \times (1-\theta_{\text{read}})} \\ &+ \frac{Size_{\text{local}}}{C_{\text{read}} \times (1-\theta_{\text{read}})} \end{aligned} \tag{12}$$

Therefore, the total execution time of Reduce $task_i$ executing on Executor e_j is:

$$\begin{aligned} Time_{i,j} = &TimeTrans_{i,j} + TimeSpill_{i,j} \\ &+ TimeCal_{i,j} + TimeRead_{i,j} \end{aligned} \tag{13}$$

B. Optimal Scheduling Problem Transformation

In Spark's Executor-Task scheduling system, the task-Executor assignment relationship is modeled as a $n \times m$ bipartite graph adjacency matrix P, where n is the number of unscheduled tasks and m is the number of idle Executors. The element $p_{i,j}$ is a binary variable: $p_{i,j}$ = 1 if $task_i$ is assigned to Executor e_j, otherwise 0.

$$P = \begin{bmatrix} p_{0,0} & \cdots & p_{0,m-1} \\ \vdots & \ddots & \vdots \\ p_{n-1,0} & \cdots & p_{n-1,m-1} \end{bmatrix} \tag{14}$$

During scheduling, the following constraints must be satisfied:

1. Task atomicity: Each task is assigned to at most one Executor.

2. Executor capacity: Each Executor runs at most one task per scheduling interval.

Mathematically, these are expressed as:

$$\begin{aligned} &\sum_{j=0}^{m-1} p_{i,j} \leq 1, \quad \forall i \in [0, n-1]; \\ &\sum_{i=0}^{n-1} p_{i,j} \leq 1, \quad \forall j \in [0, m-1] \end{aligned} \tag{15}$$

Additionally, the number of valid assignments equals the minimum of unscheduled tasks and idle Executors:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} p_{i,j} = min\{m, n\} \tag{16}$$

Meanwhile, by sensing task data volume and integrating node performance with real-time loads, the predicted execution time of tasks on different Executors is defined as an $n \times m$ matrix T using the task execution time model from the previous section:

$$T = \begin{bmatrix} Time_{0,0} & \cdots & Time_{0,m-1} \\ \vdots & \ddots & \vdots \\ Time_{n-1,0} & \cdots & Time_{n-1,m-1} \end{bmatrix} \tag{17}$$

where $Time_{i,j}$ represents the predicted execution time of $task_i$ on e_j.

Combining matrices P and T, the task scheduling problem in Spark is modeled as a weighted bipartite graph matching problem based on execution time. As shown in Figure 4, the constructed weighted bipartite graph $BG = (U, V, S)$ consists of task set U, Executor set V, and edge set S. An edge $s(i, j)$ denotes the assignment of $task_i$ to Executor e_j (corresponding to $p_{i,j}$=1), with edge weight $w(i, j)$ representing the execution time $Time_{i,j}$.

The total execution time of the scheduling scheme is the sum of weights for all matched edges, formalized as:

$$Time\,(P, T) = \sum_{i=0}^{n-1} P_i \times {T_i}^T \tag{18}$$

Among them, the row vector P_i in matrix P represents the assignment of $task_i$ to all m Executors, and the vector T_i denotes the execution time of $task_i$ on each Executor.

Using the unified graph model, the optimal task scheduling problem is transformed into a minimum weight bipartite graph matching problem. To obtain the optimal task scheduling scheme, the task scheduling matrix P should minimize the total execution time of all tasks. This objective function can be formalized as:

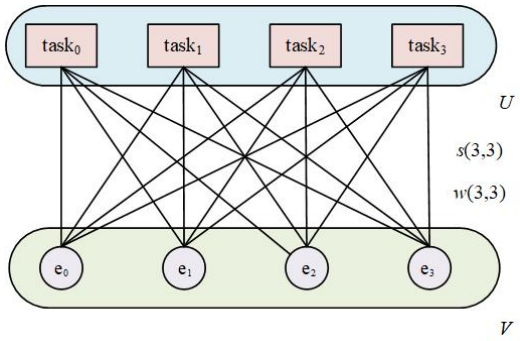

Fig. 4. Weighted Bipartite Graph Matching.

$$MinTime\,(P, T) = Min \sum_{i=0}^{n-1} P_i \times {T_i}^T \tag{19}$$

Minimizing total execution time not only reduces job completion time but also balances Executor resource loads through multidimensional load-aware weight modeling, thereby optimizing cluster resource utilization.

C. Weighted Bipartite Graph Construction Algorithm

In the specific construction of the weighted bipartite graph, edge weight calculation is first performed using task data, node performance, and real-time load statistics provided by Algorithm 1. This is combined with the execution time models of Map tasks (Eq. 8) and Reduce tasks (Eq. 13) to compute edge weights. The edge set S incorporates all possible edges and their corresponding weights to model feasible task scheduling schemes. The selection of final edges (i.e., optimal scheduling schemes) is discussed in the next subsection.

Algorithm 2 Weighted Bipartite Graph Construction Algorithm

Input: A set of TaskSet: TK; Executor assigned when scheduling a Task: SE; Partitioned data information for a Task: T_{p_info}; Node performance set: $LM\{Net_{ij}, TimeCal_{i,j}, CalSpeed_j, C_{write}, C_{read}\}$; Remote data volume for a Task: D_t; Node real-time load set: $MT\{\alpha, \beta, \theta\}$;
Output: Weighted bipartite graph:$BG = U, V, S$;
Initialize: Node set:$U \leftarrow TK, V \leftarrow ME$; Initialize edge set: $S \leftarrow \{\}$;
$m = ME.size()$,$n = TK.size()$
for $(i = 0; i < n; i++)$ **do**
  for $(j = 0; j < m; j++)$ **do**
    if $(TK\,[i]\,ismapper)$ **then**
      $w(i, j) = eq(8, Size_{total}, MT, LM)$
    else
      $w(i, j) = eq(13, T_{p\_\inf o}, D_t, MT, LM)$
    end if
    $s\,(i, j)\,.addWeight(w(i, j))$
    $S = S \cup s\,(i, j)$
  end for
end for
$BG = (U, V, S)$
return BG

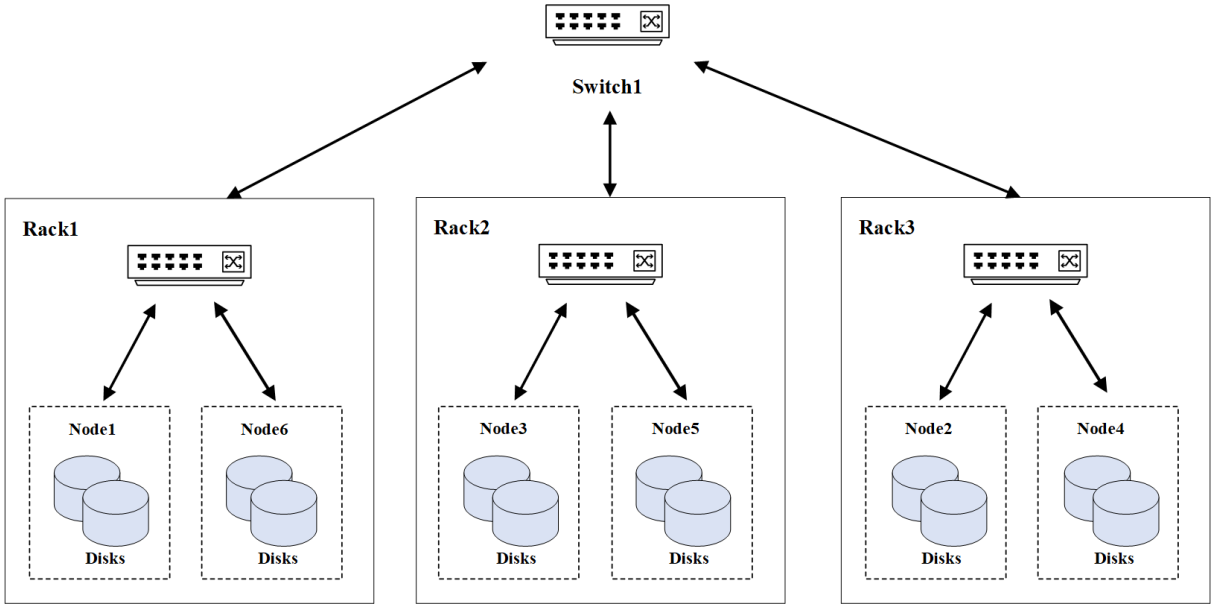

Fig. 5. Cluster Topology Diagram.

D. *Spark Task Scheduling Algorithm Based on Multidimensional Load Sensing*

In actual task scheduling, performing weight matching for all unscheduled tasks each time would lead to high time complexity. First, all tasks are sorted in descending order of execution time. Then, a fully weighted bipartite graph BG of $m \times m$ is constructed using Algorithm 2, and finally solved via the Kuhn-Munkres (KM) algorithm.

Notably, the KM algorithm is designed to solve the maximum weight matching problem, whereas our objective is to *minimize* the total execution time. To adapt our problem to the KM algorithm, we negate all edge weights $w_{i,j}$ ((i.e., the predicted execution times $Time_{i,j}$) when constructing the weight matrix. Consequently, finding the maximum weight sum becomes equivalent to finding the minimum total execution time.

Furthermore, the standard KM algorithm requires a complete bipartite graph, which means that the two sets of vertices must be of equal size. When the number of unscheduled tasks n is less than the number of idle Executors m, we create mn virtual "placeholder tasks" in the task set to meet this requirement. The edge weights from these placeholder tasks to all Executors are set to a very large positive value (which becomes a very small negative value after negation), ensuring that they will not be selected in the maximum weight matching solution and thus preserving the correct matching for real tasks.

Eventually, the algorithm returns a scheduling matrix P, which represents the optimal solution to task scheduling in this batch.

The KM algorithm solves the maximum weight matching problem for fully weighted bipartite graphs, aiming to find a matching in the bipartite graph that maximizes the sum of the weights of the matching edges. This conflicts with the objective function of minimizing the task completion time. Thus, in implementation, edge weights are set to their negative values to align the problem's objective function with the applicable conditions of the KM algorithm. Additionally, when the number of unscheduled tasks is less than the number of idle Executors (i.e., $n < m$), m–n placeholder tasks are created in BG to satisfy the KM algorithm's operational requirements. Each placeholder task has an edge weight of η (where η is greater than the weight of any edge in BG) to form the elements required for the KM algorithm to operate.

Algorithm 3 generates the optimal scheduling plan for tasks in the Map and Reduce Stages based on node performance, real-time load, the longest m tasks selected by the Longest Processing Time (LPT) strategy and the performance set of idle executors.

Algorithm 3 Spark Task Scheduling Algorithm Based on Multidimensional Load Sensing

Input: Number of tasks LPT chooses to prioritize for scheduling: m; Node real-time load set: $MT\{\alpha, \beta, \theta\}$; Assigned Executor: ME; Node performance set: $LM\{Net_{ij}, TimeCal_{i,j}, CalSpeed_j, C_{write}, C_{read}\}$
Output: Task Scheduling Matrix: P;
Initialize: Task scheduling matrix: $P \leftarrow \{\}$; Fully weighted bipartite graph: $BG \leftarrow \text{algo}(2, m, MT, LM)$;
for $w(i,j)$ in BG **do**
  $w(i,j) = -w(i,j)$
end for
$MWBM = KM(BG)$ //MWBM of BG obtained by applying KM algorithm
for $(i = 0; i < m; i++)$ **do**
  for $(j = 0; j < m; j++)$ **do**
    if $s_{(i,j)}$ is selected by the MWBM **then**
      $p_{i,j} = 1$
    else
      $p_{i,j} = 0$
    end if
  end for
end for
return P

The time complexity of generating the task scheduling

TABLE I
SPARK CLUSTER SOFTWARE CONFIGURATION

Type	Configuration
Operating system	CentOS-7.9
Hadoop version	Hadoop-2.10.2
Spark version	Spark-2.4.6
Java version	1.8
Scala version	2.12.10
Prometheus	2.26.0
Node Exporter	1.1.2

solution matrix P in Algorithm 3 is $O(m^2)$, and the time complexity of the KM algorithm is $O(m^3)$. Thus, the total time complexity is $O(m^3) + O(m^2)$. Additionally, task selection based on the LPT strategy requires sorting all tasks once, with a time complexity of $O(nlogn)$. Constructing the fully weighted bipartite graph BG using Algorithm 2 has a time complexity of $O(m^2)$. In practical scenarios, the number of Executors assigned (m) is typically much smaller than the number of tasks (n). Therefore, the overall time complexity of the algorithm is dominated by $O(nlogn)$ due to $m << n$.

IV. EXPERIMENT

In this paper, we extend the task scheduling module of Spark 2.4.6 via the TaskSchedulerImpl interface, embedding the MDLS strategy into the native framework as a plugin to ensure compatibility with the existing cluster task execution mechanism. We conduct comparative experiments in a heterogeneous cluster environment.

A. Experimental Setup

The experimental cluster consists of six physical servers, including one master node and five worker nodes (Note: "slave" is updated to "worker" for modern terminology consistency). Each node is a multi-role node deployed with other service applications (Note: "applications" is more technically precise than "programs"). We use Prometheus to monitor real-time resource usage of each node in the cluster. Nodes in the cluster share the same software environment, and specific version information is shown in Table I.

The servers are distributed across three different racks. Each rack contains two nodes, and the nodes in the racks are interconnected via a single switch, with network bandwidths of 10Gbps, 500Mbps, and 1Gbps for the three racks, respectively. The cluster topology is shown in Figure 5. The distribution of cluster nodes and hardware configuration details are listed in Table II.

TABLE II
CLUSTER NODE DISTRIBUTION AND HARDWARE CONFIGURATION

Cluster Information	Rack1		Rack2		Rack3	
	Node1	Node6	Node3	Node5	Node2	Node4
CPU Frequency (GHZ)	2.1	2.2	2.2	2.4	2.1	2.4
Number of Allocated CPU Cores	6	6	6	6	6	6
Allocated Memory (GB)	48	48	48	48	48	48
Network Bandwidth (Bps)	10G	500M	500M	500M	1G	1G
Disk Capacity (TB)	83	30	30	2	83	2
Disk Write Performance	700m/s	767m/s	1.5g/s	515m/s	670m/s	536m/s
Disk Read Performance	2.3g/s	3g/s	2.5g/s	829m/s	670m/s	835m/s
Is Hadoop Node	Yes	Yes	Yes	No	Yes	No

B. Single-Round Performance Analysis

Single-round jobs, typically comprising one Map stage and one Reduce stage, are a key benchmark for evaluating a scheduler's fundamental efficiency. We demonstrate the superiority of MDLS by analyzing three different types of benchmarks.

WordCount, a widely used task in large-scale data processing, aims to count the occurrences of each word in the input data. While the shuffle data volume in WordCount is relatively small, its performance is still sensitive to computation and network transfer efficiency. As shown in Figure 6, MDLS maintains a performance lead on the 300GB dataset. This is primarily attributed to its global decision-making capability based on optimal matching. Within a scheduling batch, MDLS does not assign tasks one-by-one using a greedy approach like DEF or RUPAM. Instead, it solves for a batch-level "Pareto optimal" solution using the KM algorithm. This means it can balance the computational needs and data locations for multiple tasks simultaneously, minimizing the total completion time for all tasks in that batch and thus accumulating small but critical advantages at every scheduling decision point.

Experimental data show that for a 200GB dataset, MDLS reduces job execution time by 6.09% compared with the DEF scheduling strategy and by 3.24% compared with RUPAM. As dataset size increases, MDLS' performance advantage becomes more pronounced: for a 250GB dataset, job execution time is reduced by 6.8% compared with DEF and 3.4% compared with RUPAM; for a 300GB dataset, execution time is reduced by 7.43% compared with DEF and 4.5% compared with RUPAM. As dataset size grows, data transmission and processing pressure increase. MDLS optimizes bandwidth allocation to avoid network congestion under high load and enhances network utilization under low load, enabling synergy between application-layer transmission semantics and network-layer resource conditions. This effectively reduces job execution time across datasets of different sizes.

Sort [33], a common workload benchmark task primarily used for sorting data objects, has experimental results presented in Figure 7. Experimental data show that MDLS demonstrates significant performance advantages across datasets of varying sizes.

For a 40GB dataset, MDLS reduces execution time by 9.3% compared with DEF and 3% compared with RUPAM. For a 60GB dataset, MDLS further shortens execution time by 10.5% compared with DEF and 3.7% compared with RUPAM. MDLS performs even better for an 80GB dataset, with execution time reduced by 11% compared with DEF and 2.7% compared with RUPAM. Although the RUPAM strategy employs a static priority queue for task resource allocation, it cannot adjust in real time according to data size and task load changes, limiting its performance improvement. In contrast, MDLS uses a multi-dimensional load-aware weight allocation mechanism to monitor task load in real time and dynamically adjust task allocation weights based on multi-dimensional factors such as data size and computational complexity. In the Sort task, as data size increases, task load and complexity also increase. MDLS adapts better to these changes by allocating resources reasonably to avoid computational resource imbalance, thereby significantly improving the execution efficiency of the Sort task.

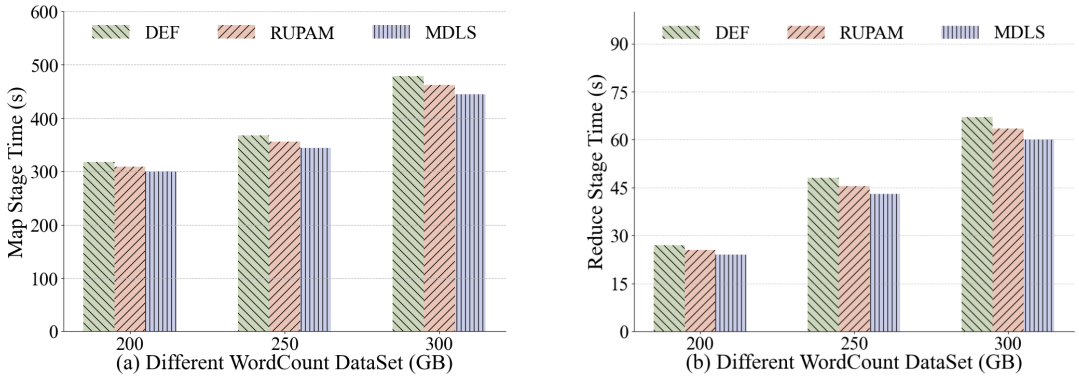

Fig. 6. Execution Results of the WordCount Benchmark.

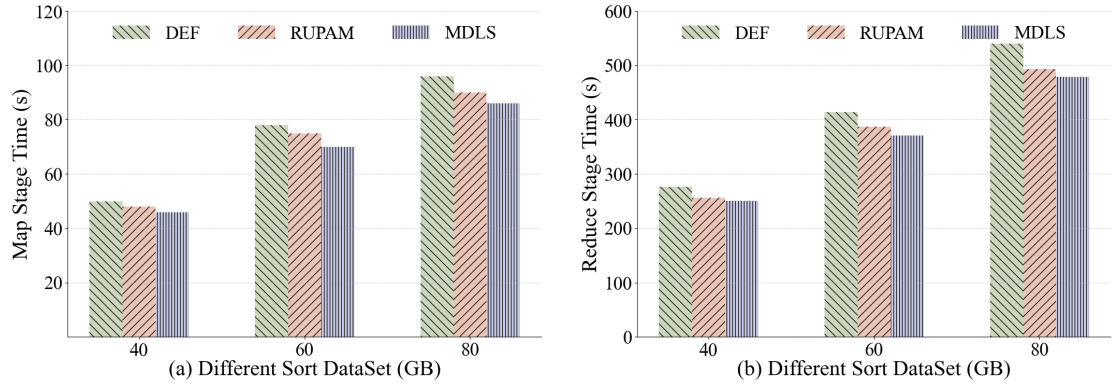

Fig. 7. Execution Results of the Sort Benchmark.

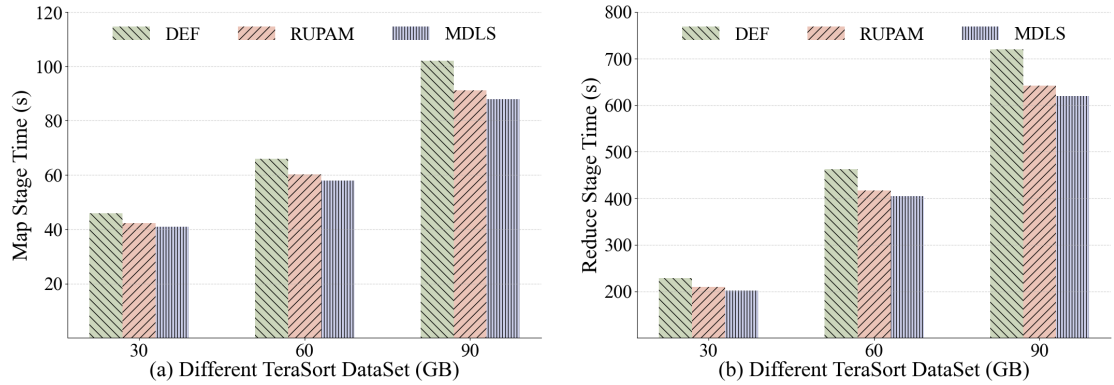

Fig. 8. Execution Results of the TeraSort Benchmark.

TeraSort [34], a sorting and merging framework based on MapReduce technology, is designed to handle large-scale data sorting and merging. Its test suite is provided by HiBench, with experimental results summarized in fig8. Across the three data sizes, MDLS outperforms DEF by 11.3%–13.9%, while the RUPAM strategy achieves an 8.2%–10.9% improvement. The DEF strategy lacks comprehensive consideration of multidimensional resources in large-scale data processing, failing to schedule tasks reasonably based on data distribution and node load. This leads to low resource utilization and consequently long task execution time. By contrast, MDLS assigns tasks to different nodes according to data distribution, avoiding concentration and waste of computing resources. This significantly optimizes the scheduling and execution efficiency of large-scale data sorting and merging.

C. Iterative Job Performance Analysis

The above three workloads each consist of only two Stages and one Shuffle. However, real-world big data processing scenarios often involve more complex task flows. To comprehensively demonstrate the performance advantages of the multidimensional load-aware task scheduling strategy (MDLS), three workloads with more Stages—Linear Regression (LR), PageRank (PR), and Random Forest (RF)—were

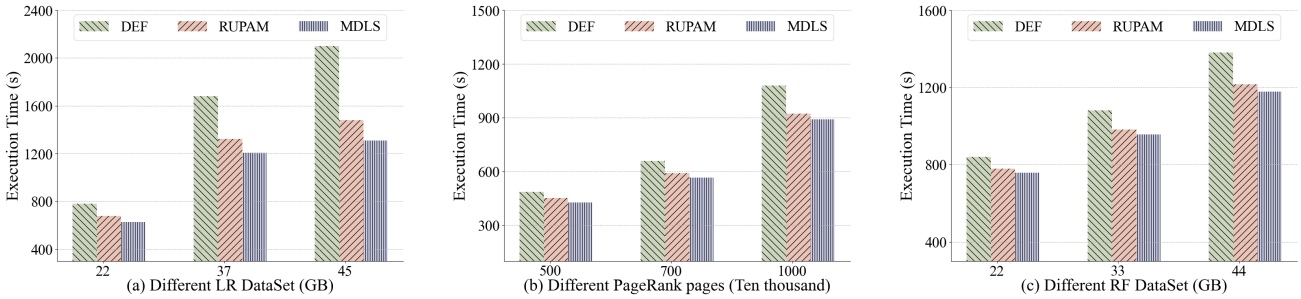

Fig. 9. Execution Results of the Multi-Stage Benchmark.

selected, and experiments were conducted on various datasets of different sizes. Experimental results are shown in Figure 9.

As shown in Figure 9, MDLS achieves the most significant performance improvements across all iterative jobs, with gains of up to 37.6%. This powerfully demonstrates the Cumulative Advantage of the MDLS strategy. The greedy or static approaches of DEF and RUPAM may make locally optimal but globally suboptimal decisions at each stage, accumulating "scheduling debt." MDLS, by striving for a global optimum in every scheduling batch, effectively prevents this negative compounding effect. For example, across the multiple iterations of PageRank, MDLS consistently assigns computation tasks to nodes with lower CPU loads while preserving network bandwidth for subsequent data shuffles. This forward-looking, cross-stage resource balancing is a capability that the other two strategies lack and is the fundamental reason for MDLS's success in complex job flows.

D. Resource Load Analysis

To further demonstrate MDLS advantages, we collected samples of CPU and network resource utilization during runtime for three typical workloads: PageRank, Sort, and WordCount. Resource utilization results are shown in Figure 10 and Figure 11. Experimental results indicate that while DEF enhances data locality through delay scheduling to reduce application-layer data transmission, its lack of network-layer bandwidth and load awareness leads to delays from network congestion or underutilization. For example, during peak data transmission, DEF cannot adjust task allocation in time, causing overutilized network bandwidth that impacts other tasks and increases execution time. RUPAM's static queuing mechanism also fails to dynamically sense network load changes or adjust task allocation based on real-time bandwidth conditions, resulting in inefficient network resource use. In contrast, MDLS dynamically adjusts task allocation weights under high network load by monitoring multidimensional resource usage, controlling bandwidth to avoid congestion while enhancing utilization under low load. This creates synergy between application-layer transmission and network-layer resources, balancing resource load while minimizing task completion time.

In terms of CPU resource utilization, as shown in Figure 11, MDLS exhibits significantly higher CPU utilization than the other two methods across three typical workloads. As shown in Figure 10, MDLS achieves significantly higher CPU utilization than its counterparts in all tests. The box plot for the Sort job, in particular, reveals the superior load balancing capability of MDLS: it achieves the highest median CPU utilization while also displaying the narrowest interquartile range (IQR). This demonstrates that CPU utilization under MDLS is not only higher overall but also more uniform across the nodes. Such a distribution provides empirical evidence of MDLS's success in mitigating load imbalance by migrating tasks from contended to available nodes, which in turn improves the cluster's parallel efficiency. In the PageRank job, MDLS reaches a peak CPU utilization of 51.8%, compared to DEF's 46.2%; in the Sort job, MDLS achieves a peak CPU utilization of 41.5%, significantly higher than DEF's 29.85%; in the WordCount job, MDLS's peak CPU utilization is 40.1%, surpassing DEF's 33.65% and RUPAM's 36.45%.

The DEF strategy lacks the ability to dynamically allocate CPU resources, failing to distribute CPU resources reasonably according to task computational demands. This leads to idle CPU resources on some nodes and overuse on others, reducing overall CPU utilization. RUPAM's static queuing mechanism fails to adjust task allocation in real time to adapt to CPU load changes, leading to inefficient CPU resource utilization. By contrast, MDLS pre-senses data distribution and task characteristics, dynamically adjusting allocation weights based on real-time load of different resource types at each node. This avoids severe computing resource skewing, balances node resource loads, and minimizes completion time, demonstrating its load-sensitive elastic scheduling capability.

In summary, compared with Spark's default scheduling, the RUPAM strategy uses static priority queues for resource types (e.g., CPU, network, storage, and memory), sorting them by capacity in descending order and allocating tasks via feature-queue matching to optimize execution efficiency. However, its static queuing mechanism cannot dynamically sense Executor real-time load changes, disconnecting task allocation from current node load states. Similarly, some studies optimize from the perspective of task cost and latency benefits, such as the CALS strategy proposed by Xu et al. [35], which uses a greedy algorithm to optimize task execution order and dynamically adjusts delay times. In contrast, the MDLS strategy in this paper goes a step further. It first employs the LPT strategy for batch task selection, then constructs a multi-dimensional load-aware task execution time model by integrating node computing capability, data distribution, and Executor real-time load. It builds a fully weighted bipartite graph and solves for the

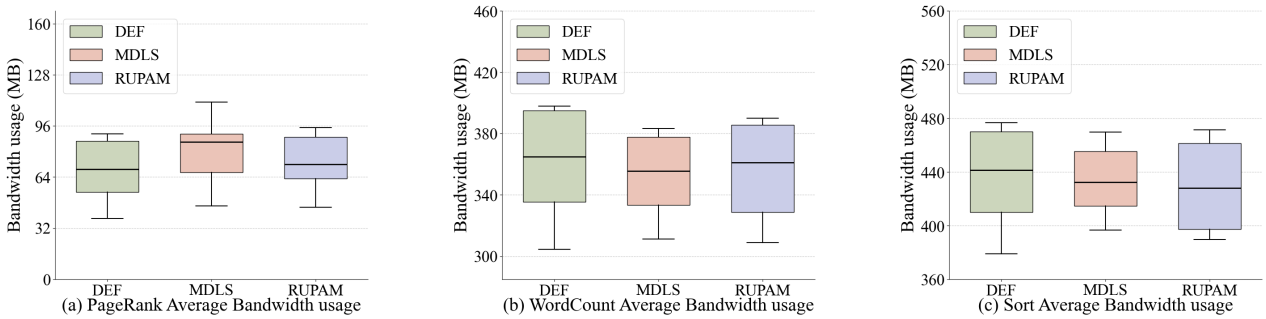

Fig. 10. Average BandWidth Usage For Multiple Loads

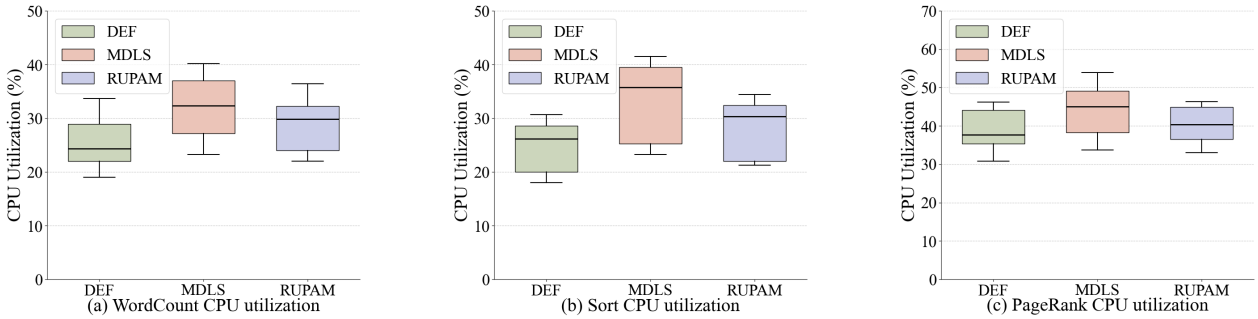

Fig. 11. CPU Utilization For multiple Loads

batch-level global optimal scheduling scheme via the KM algorithm. This reduces task execution time while avoiding computing resource load skewing.

Experimental results indicate that although MDLS introduces moderate scheduling delays through real-time resource monitoring and heuristic decision-making, it achieves significant performance gains compared to Spark's default scheduling.

V. Conclusion

Cluster performance and load heterogeneity are critical factors affecting Spark task scheduling. This paper proposes a Spark task scheduling strategy based on Multidimensional Load Sensing (MDLS). The strategy first assesses the performance and real-time load of each compute node, counts the data load and distribution of compute tasks, and performs multi-dimensional dynamic modeling of task execution times on different Executors. It then analyzes Spark's Executor-Task scheduling mechanism in depth, transforming the optimal task scheduling problem into a minimum weighted bipartite graph matching problem. Finally, combining Spark's task execution mechanism with the Longest Processing Time (LPT) strategy, it constructs a fully weighted bipartite graph and solves for the batch optimal scheduling plan using the Kuhn-Munkres (KM) algorithm. MDLS addresses the performance bottlenecks of Spark's default scheduling policy, which focuses solely on data locality and uses a polling-based load balancing approach in heterogeneous clusters. Experimental results show that MDLS improves Spark cluster job execution efficiency, balances node load more effectively, and achieves up to 37.6% performance improvement.

Data and Code Availability

The data and code supporting this study are available in the GitHub repository at https://github.com/wcy666103/spark-paper-task

References

[1] X. Xu, W. Dou, X. Zhang, and J. Chen, "EnReal: An Energy-Aware Resource Allocation Method for Scientific Workflow Executions in Cloud Environment," IEEE Transactions on Cloud Computing, vol. 4, no. 2, pp. 166–179, 2016.

[2] Z. Liu and T. S. E. Ng, "Leaky Buffer: A Novel Abstraction for Relieving Memory Pressure from Cluster Data Processing Frameworks," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 1, pp. 128–140, 2017.

[3] H. Moniz, J. Leitão, R. J. Dias, J. Gehrke, N. Preguiça, and R. Rodrigues, "Blotter: Low Latency Transactions for Geo-Replicated Storage," Proceedings of the 26th International Conference on World Wide Web, pp. 263–272, 2017.

[4] Q. Gan, X. Wang, and X. Fang, "Efficient and Secure Auditing Scheme for Outsourced Big Data with Dynamicity in Cloud," Science China Information Sciences, vol. 61, pp. 122104, 2018.

[5] K. Wang, C. Xu, Y. Zhang, S. Guo, and A. Y. Zomaya, "Robust Big Data Analytics for Electricity Price Forecasting in the Smart Grid," IEEE Transactions on Big Data, vol. 5, no. 1, pp. 34–45, 2019.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Commun. ACM, vol. 51, no. 1, pp. 107–113, 2008.

[7] S. M. Nabavinejad, M. Goudarzi, and S. Mozaffari, "The Memory Challenge in Reduce Phase of MapReduce Applications," IEEE Transactions on Big Data, vol. 2, no. 4, pp. 380–386, 2016.

[8] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "MapTask Scheduling in MapReduce with Data Locality: Throughput and Heavy-Traffic Optimality," IEEE/ACM Trans. Netw., vol. 24, no. 1, pp. 190-203, 2016.

[9] Apache Spark, 2019.[Online].Available: http://spark.apache.org/

[10] T. Hajji, R. Loukili, I. El Hassani, and T. Masrour, "Optimizations of Distributed Computing Processes on Apache Spark Platform," IAENG International Journal of Computer Science, vol. 50, no. 2, pp422-433, 2023.

[11] S. N. Naik, A. Negi, B. R. Tapas Bapu, and R. Anitha, "A data locality based scheduler to enhance MapReduce performance in heterogeneous environments," Future Generation Computer Systems, vol. 90, no. 1, pp. 423-434, 2019.

[12] M. Tang, C. Wang, and Y. Peng, "MARS: Scheduling non-local tasks in mapreduce," 2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems, pp. 536-540, 2014.

[13] D. Choi, M. Jeon, N. Kim, and B.-D. Lee, "An Enhanced Data-Locality-Aware Task Scheduling Algorithm for Hadoop Applications," *IEEE Systems Journal*, vol. 12, no. 4, pp. 3346–3357, Dec. 2018.

[14] F. Shang, X. Chen, and C. Yan, "A strategy for scheduling reduce task based on intermediate data locality of the MapReduce," Cluster Computing, vol. 20, no. 4, pp. 2821-2831, 2017.

[15] Z. Fu, M. He, Y. Yi, and Z. Tang, "Improving Data Locality of Tasks by Executor Allocation in Spark Computing Environment," IEEE Trans. Cloud Comput., vol. 12, no. 3, pp. 876-888, 2024.

[16] L. Xu, A. R. Butt, S.-H. Lim, and R. Kannan, "A heterogeneity-aware task scheduler for spark," 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 245-256, 2018.

[17] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing," 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Newport Beach, CA, USA, pp. 295-304, 2011.

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," 2009 ACM SIGMOD International Conference on Management of Data, Providence, Rhode Island, USA, pp. 165-178, 2009.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleey, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," EuroSys, Paris, France, 2010.

[20] Z. Guo, G. Fox, and M. Zhou, "Investigation of Data Locality in MapReduce," 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012), Ottawa, ON, Canada, pp. 419-426, 2012.

[21] S. Lu, M. Zhao, C. Li, Q. Du, and Y. Luo, "Time-Aware Data Partition Optimization and Heterogeneous Task Scheduling Strategies in Spark Clusters," The Computer Journal, vol. 67, no. 2, pp. 762-776, 2024.

[22] Z. Tang, A. Zeng, X. Zhang, L. Yang, and K. Li, "Dynamic memory-aware scheduling in spark computing environment," Journal of Parallel and Distributed Computing, vol. 141, pp. 10–22, 2020.

[23] H. Li, H. Wang, S. Fang, Y. Zou, and W. Tian, "An energy-aware scheduling algorithm for big data applications in Spark," Cluster Comput, vol. 23, pp. 593–609, 2020.

[24] Z. Tang, Z. Xiao, L. Yang, K. He, and K. Li, "A Network Load Perception Based Task Scheduler for Parallel Distributed Data Processing Systems," IEEE Transactions on Cloud Computing, vol. 11, no. 2, pp. 1352–1364, 2023.

[25] Y. Zhang, C. Wang, X. He, J. Yu, R. Zhai, and Y. Song, "Delay-aware resource-efficient interleaved task scheduling strategy in spark," Computer Science and Information Systems, pp. 18–18, 2025.

[26] G. Wang, Y. Wang, M. S. Obaidat, C. Lin, and H. Guo, "Dynamic Multiworkflow Deadline and Budget Constrained Scheduling in Heterogeneous Distributed Systems," IEEE Systems Journal, vol. 15, no. 4, pp. 4939–4949, 2021.

[27] Y. Wang, R. R. Yang, Y. X. Xu, X. Li, and J. L. Shi, "Research on Multi-Agent Task Optimization and Scheduling Based on Improved Ant Colony Algorithm," IOP Conference Series: Materials Science and Engineering, vol. 1043, no. 3, pp. 032007, 2021.

[28] G. Singh, A. Sharma, R. Jeyaraj, and A. Paul, "Handling Non-Local Executions to Improve MapReduce Performance Using Ant Colony Optimization," IEEE Access, vol. 9, pp. 96176-96188, 2021.

[29] H. Gu, X. Li, and Z. Lu, "Scheduling Spark Tasks With Data Skew and Deadline Constraints," IEEE Access, vol. 9, pp. 2793-2804, 2021.

[30] L. -F. Neciu, F. Pop, E. -S. Apostol, and C. -O. Truică, "Efficient Real-time Earliest Deadline First based scheduling for Apache Spark," 2021 20th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 97-104, 2021.

[31] Y. Zhao, J. Dong, H. Liu, J. Wu, and Y. Liu, "Performance Improvement of DAG-Aware Task Scheduling Algorithms with Efficient Cache Management in Spark," Electronics, vol. 10, pp. 1874, 2021.

[32] W. CongYang, Y. JunYang, and L. Han, "Dynamic-awareness-based Spark Execution Plan Selection and Shuffle Optimization Strategies," IAENG International Journal of Computer Science, vol. 52, no. 7, pp2223-2233, 2025

[33] R. Baity, L. R. Humphrey, and K. Hopkinson, "Formal Verification of a Merge Sort Algorithm in SPARK," AIAA Scitech 2021 Forum, Reston, AIAA, p. 0039, 2021.

[34] D. Xia, M. Simpson, V. Srinivasan, and A. Thomo, "Strongly Minimal MapReduce Algorithms: A TeraSort Case Study," in Foundations of Information and Knowledge Systems (FoIKS 2020), Lecture Notes in Computer Science, vol. 12012, Springer, Cham, pp. 301–317, 2020.

[35] Q. Xu, C. Wang, and J. Yu, "A Cost-Aware and Latency-Benefit Evaluation-Based Task Scheduling Optimization Strategy in Apache Spark," Concurr. Comput. : Pract. Exper., 2025.

Congyang Wang born in 2000, received a bachelor's degree in software engineering from Henan University in 2022, where he graduated from the School of Software. He is currently a graduate student in the School of Software at Henan University. His research interests include distributed computing and big data processing.He has published three SCI journal paper and won several national awards in computer-related competitions. He is a CCF (China Computer Federation) student member and an Alibaba Cloud Open Source Pioneer. Since 2022, he has completed two industrial projects and holds two patents. As an Apache Contributor, he has made technical contributions to open-source projects in the big data field.