**Computer Science and Application** 计算机科学与应用, 2016, 6(11), 692-704
Published Online November 2016 in Hans. http://www.hanspub.org/journal/csa
http://dx.doi.org/10.12677/csa.2016.611084

Hans汉斯

# Adaptive Scheduling Strategy for Heterogeneous Spark Cluster

**Jiajun Xu, Gongshen Liu, Bo Su, Kui Meng**

Shanghai Jiao Tong University, Shanghai
Email: 414369805@qq.com, lgshen@sjtu.edu.cn, subo@sjtu.edu.cn, mengkui@sjtu.edu.cn

Received: Nov. 3rd, 2016; accepted: Nov. 21st, 2016; published: Nov. 24th, 2016



## Abstract

**The scheduling strategy of Spark assumes that cluster is homogenized. However, as the change or update of hardware in cluster, it becomes more and more heterogeneous. Thus, the original scheduling strategy cannot meet the performance requirement anymore and short board effect gradually emerges. The paper proposes a new strategy to solve this problem. The new strategy refers the idea of hierarchical scheduling. It combines the task complexity, worker performance and worker CPU usage as its scheduling factors to improve the scheduling performance. And experiments show that the new strategy is absolutely effective.**

## Keywords

**Spark, Heterogeneous Cluster, Scheduling Strategy**

# 基于Spark的异构集群调度策略研究

**徐佳俊，刘功申，苏　波，孟　魁**

上海交通大学，上海
Email: 414369805@qq.com, lgshen@sjtu.edu.cn, subo@sjtu.edu.cn, mengkui@sjtu.edu.cn

收稿日期：2016年11月3日；录用日期：2016年11月21日；发布日期：2016年11月24日

文章引用: 徐佳俊, 刘功申, 苏波, 孟魁. 基于 Spark 的异构集群调度策略研究[J]. 计算机科学与应用, 2016, 6(11): 692-704. http://dx.doi.org/10.12677/csa.2016.611084

## 摘　要

**Spark的原生调度策略建立在集群同质化的基本假设上。然而随着硬件的更迭以及高性能硬件的引入，集群异质化现象日趋显著。因此现有的调度策略在异构集群环境下并不高效，短板效应严重。针对这个问题，本文提出了一种新的调度策略以优化Spark在异构集群下的表现。新策略引入了分层调度的思想，调度时综合考量了任务复杂度、节点性能及节点资源使用情况等因素，实现了更加高效公平的任务调度算法。通过仿真和真机实验，证明了新策略的效果相对于原策略有明显提升。**



## 关键词

**Spark，异构集群，调度策略**


## 1. 引言

随着大数据时代的到来，几乎所有行业都开始面临着大数据处理带来的挑战，海量数据的分析处理已经无法继续在单一节点的机器上完成，即便计算机性能按照摩尔定律已经越来越高，但依然无法满足需求。因此如何构建大数据的分布式计算平台成为了基础但也极具挑战性的工作。

为摆脱单一节点的瓶颈限制，一般采用分布式并行的计算框架搭建集群。其中 MapReduce [1]是目前最广为采纳的一种，基于该框架，Apache 于 2005 年实现了 Hadoop [2]系统，并通过 Hive 和 HDFS 提高了分布式数据库和文件系统访问效率。由于 Hadoop 的实现基于文件读写，所以在处理迭代性强、I/O 密集型的任务时，存在致命的瓶颈，而且主要通过冗余实现容错，效率较低。因此在 2009 年，Spark [3]系统在加州大学伯克利分校 AMP 实验室应运而生，首创的 RDD [4]存储结构实现了被处理数据的内存直接访问，极大提高了访问效率也避免了频繁的文件 I/O 交互，同时也采用 Lineage 机制，保证了系统的容错性。另外还对 Hive、HDFS 兼容，并且提供了 RDD 数据的用户级处理接口，极大提高了系统的灵活性。因此，Spark 已经成为当下主流的分布式计算系统。

然而，无论是 Hadoop 还是 Spark，都基于一个理想化假设，即集群节点同构性的存在。然而实际集群环境下，由于数据中心硬件资源的更迭、GPU 等高性能计算组件的引入等原因，集群异构性[5]问题广泛存在。另外，任务本身也存在异构性问题，计算密集型和 I/O 密集型任务对计算资源需求不一致，分配节点时也应考虑。而目前 Spark 平台在做任务调度时，并不考虑集群异构性及节点资源利用情况，则集群中各节点在任务执行时会出现负载不均衡的问题，导致一部分节点资源消耗过载，另一部分冗余的问题。由于短板效应，整体效率将受制于集群的弱节点，导致整体性能的下降。

因此本文基于 Spark，提出了一种新的异构集群调度策略，来解决上述问题，以期提高整体执行效率。

章节 2 将分析目前 Spark 如何实现任务调度，并对现有的一些针对分布式集群异构性问题的解决方案进行总结并给出本文的新思路；章节 3 将详述本文的调度策略思路；章节 4 将通过实验验证该策略的可行性和效果；章节 5 将总结并给出一些后续思考。

## 2. 相关研究及创新

### 2.1. 现有方案分析

由于 Hadoop 技术更加成熟且在企业内使用更广且更久，因此目前针对分布式计算平台集群异构问题的优化方案主要集中在 Hadoop 平台，如：Facebook, Twitter, Amazon, Baidu 等。Yahoo 和 Facebook 也在

调度策略方向做过研究，分别提出了 Capacity Scheduler [6]和 FAIR Scheduler，均已被 Apache 加入到 Hadoop 中。2009 年，Matei Zaharia [7] [8]提出了 Delay Scheduling 的策略，并开发了 FAIR Scheduler，解决了多用户资源分配公平性问题和单队列阻塞问题，Spark FAIR Scheduler 的思路即来源于此。2005 年，Edmund B. Nightingale [9]提出了 Speculative Task Execution Strategy (STES)，其思路是当节点空闲时静默预测执行其他节点在本地的任务副本，提高节点利用率，通过执行前保存快照，确保预测失败可回滚。2008 年，Matei Zaharia [10]改进了 STES，提出了 Improved STES，用任务的预计完成时间替代任务完成进度作为调度依据。2009 年，Mark Yong [11]提出了 Task Tracker Resource Aware Scheduler (TRAS)，因为即便是 Improved STES 也会因为副本分布不均导致资源分配不均，即可预测执行的任务集中在忙碌节点附近。TRAS 在 Hadoop 的任务跟踪器(Task Tracker)中引入资源监控模块，记录节点资源使用情况，然后在作业跟踪器(Job Tracker)中加入了两种调度策略，一种是依据监控数据动态分配每个 Task Tracker 的执行任务数；另一种是固定执行 Task Tracker 的任务数，将所有可用执行节点存放于队列中用于调度。该策略考虑了资源使用情况，并利用延迟调度保证了资源分配的相对公平。2013 年，Zhuo Tang [12]提出了 Map Reduce Task Scheduler for Deadline (MTSD), MTSD 首先将异构集群中的所有节点根据计算能力划分为多个层级，再针对不同级别节点估算其上任务的预计完成时间，区分 Map 任务和 Reduce 任务，然后根据完成时间和任务截止时间作为依据调度。2014 年，Xiaolong Xu [13]提出了 Adaptive Task Scheduling on Dynamic Workload Adjustment(ATSDWA)，ATSDWA 在每个节点部署了一个监测模块，收集节点资源使用情况和执行时间。每个节点根据硬件配置初始化最大任务数，并由任务调度器(Task Scheduler)在每个心跳包传输时反馈，然后依据监测信息动态调整各节点最大可运行任务数，从而实现了 Hadoop 的动态自适应调度。

Spark 方面，2016 年，Zhiwei Yang 提出了 Heterogeneous Spark Adaptive Task Scheduling (HSATS)，其思路与 ATSDWA 类似，在 Worker 节点加入监测模块，并在 Task Scheduler 中加入节点权值动态调整算法，根据节点的实时权值完成调度。

综上所述，异构集群的调度优化问题的解决思路演化大致是从单队列到多队列、静态分配到动态调节、按序执行到预测执行，另外在此基础上，调度反馈、剩余时间估计、异构集群分区等思路也不断在被挖掘和优化。不过更多研究成果依然集中在 Hadoop 上，随着 Spark 的崛起，企业也将慢慢由 Hadoop 向 Spark 转型，相信更多的研究也将随之转向 Spark，不过 Hadoop 的研究及成果仍有重要的参考意义。

## 2.2. 新思路

前文概述了异构集群调度优化的既有解决方案，包括 Hadoop 和 Spark。下面给出一种新的思路。

之前的解决思路更多关注的是如何实现异构集群资源的平均分配，即希望通过调度策略使异构集群在资源利用层面上可视为同构。而本文希望在此基础上，再充分利用集群异构性，以提高集群效率。

存储体系中，常通过多级高速缓存提高整体系统的效率，高速缓存的特点即单点效率高但容量较小。这里就是利用了存储介质的异构性，通过引入高性能的异构介质实现了整体系统的效率提升。同理，目前的分布式集群仍处于单层结构，而计算元件其实也类似于存储元件，其性能在不断提高，所以单层的集群结构是有待优化的。所以笔者的异构集群优化思路就是通过将集群进行分层，实现整体效率提升。其在实际应用场景的意义在于，可以通过向集群中引入少量高速元件作为高性能节点，从而提高整体分布式系统的效率。而该策略关键要解决的问题就在于如何将集群分层并将不同复杂度的任务按需分配到各个层级节点上。

因此本文的调度策略的主要思路即在现有的动态调节基础上，引入分层调度思想，以期实现整体性能的提升。

## 3. 调度架构及策略

### 3.1. 任务识别

Spark 在提交作业(Job)后，会将其划分为多个阶段(Stage)，并根据之间的依赖关系进一步切分为多个任务(Task), Task 是最终调度的基本单元。这一节将从微观的角度分析任务识别的可行性。

首先，当所提交程序调用 Spark 的 Transformation 或 Action 操作接口时，SparkContext 会对传入的闭包进行序列化检查，保证其可序列化传输并反序列化执行。清理好的闭包函数将作为初始化参数写入创建的 RDD 中。此时，Spark 即将 RDD 数据和闭包处理函数绑定，由于 RDD 是 Spark 的数据流主线，所以中间无论是执行 Stage 划分调度还是 Task 划分调度，都不影响 RDD 与其闭包的绑定关系。当最终分配到 Worker 节点上的 Executor 开始实际执行时，会解析并调用 RDD 上绑定的闭包执行函数对数据进行迭代操作。

所以从微观角度分析，闭包操作是可以追溯的，因此可以通过在 RDD 创建时，对传入的闭包进行复杂度分析，得到闭包复杂度并写入 RDD，作为最终调度时的参考指标。

**定义一：(任务复杂度)** 定义 $C_p\left(p=1,2,\cdots,M\right)$ 为每个任务的复杂度，$M$ 为任务总数。调度策略将依据任务的复杂度决定调度的优先级。

### 3.2. 集群分层

集群异构性随着硬件更迭和高性能硬件引入将趋于显著，目前 Spark 调度仅依据 CPU 核数实现调度，该假设对实际的异构集群并不“公平”。一方面，单一 CPU 核心的计算能力存在明显差异；另一方面，GPU 等高计算性能硬件的计算能力远高于 CPU，以核心数进行分配，实际上忽略了每个核心本身的异构性，调度粒度不够细。

所以本文的调度方案首先将集群根据核心的性能进行分层，作为最终调度策略的指标之一。基本分层思路是在集群的不同节点上取单一核心，在其上分别运行多个benchmark作业并记录相应的执行耗时，根据指定的分层数进行划分。

**定义二：(节点性能指数)** 定义 $P_i\left(i=1,2,\cdots,N\right)$ 为节点性能指数，$N$ 为节点个数，该指数大小代表对应节点的 CPU 计算性能，指数越大，说明该 CPU 性能越强。

**定义三：(节点分层级别)** 定义 $K$ 为集群分层数，$L_j$ 为节点计算性能的级别，$L_j$ 的定义如下：

$$L_j=\begin{cases}L_{j-1}\times\alpha, & 1<j<K\\ 1, & j=0\end{cases} \tag{1}$$

其中，公式中的 $\alpha$ 是一个大于 1 的常数，定义为分层指数，这样最终分层结果服从以分层指数为底的指数函数，即越高级别的节点越少，而低级别的节点较多，符合分层的期望。

根据上述两个定义，下面给出集群节点的分层算法。

**算法一 (节点分层算法)**

```
T[]// benchmark 任务完成时间, size = N
L[]//节点层级信息, size = K
foreach Worker i
    运行 benchmark 任务
    T[i] =  执行时间
end
T[]按升序排序
```

```
T[]根据 T[1]归一化
L[1].add(Worker 1)
for (j = 2 to T.size)
    tmp = 1
    level = 0
    while (tmp< T[j])
        tmp *= α
        level++
    end
    level = min(level, K)
    L[level].add(Worker j)
end
return L[]
```

算法的基本思路如下：

1. 首先遍历 Cluster 中的所有 Worker，并分别运行同一组 benchmark 任务，统计每个 Worker 的执行时间；
2. 将得到的结果按升序排序，即约靠前的节点，执行时间越短，相对性能越强；
3. 以最短执行时间结果为标准，对数组初始化，用以后续评估；
4. 先将第一个节点归为$L_1$节点，然后依次遍历，并根据定义三得到所有节点对应的分层结果。

根据上述算法，将得到各层级的节点集合，然后可为对应$L_j$的节点定义其$P_i$，即相同级别的节点的性能指数相等，依据级别划分性能指数的主要目的是排除偶然性。另外，可将算法一中运行的 benchmark 进行多次更换得到多个分级结果，得到分级矩阵G，行代表不同节点，列代表不同 benchmark，最终取节点在所有 benchmark 中分级的众数作为节点最终分级结果。

### 3.3. 节点检测

前文提及 Spark 任务调度依据核心数作为分配依据，但根据核心数分配是一种静态思路，核心本身存在忙闲状态，对于在忙的 CPU 也存在其上队列长度的不同，因此需要动态对 CPU 使用情况及其上任务队列进行监控，方可实现任务的动态调度优化。

Spark 集群是主从式结构，Master 和 Worker 之间利用 RPC 机制实现通讯，因此可在每个 Worker 节点上添加一个检测模块，因为要保证 Master 和 Worker 的连接，之间会存在心跳(Heartbeat)信息交互，我们可以利用每次心跳信息交互的契机，同步一次当前节点的资源使用情况，Master 在本地保存一份 Worker 节点的资源利用表，作为下一次资源分配时的依据，以实现资源的动态分配。

具体检测的节点参数主要有：队列长度、CPU 使用率、节点性能指数三种。单节点上队列长度将影响该节点的完成时间，一般队列越长，其上的预期执行时间越长，则应该尽可能减少其上任务的分配，另外 Spark 执行时存在依赖，若队列过长，还可能造成其他节点任务依赖于队列中某个任务，导致整个作业的阻塞，因此要检测队列长度从而在调度时避免长队列的出现。而 CPU 使用率直接反映了节点核心的忙闲，CPU 占用过高，说明节点可用的计算资源匮乏，即再分配任务可能需要等待计算资源的释放，造成延迟，所以调度时应可能平衡各节点的 CPU 占用。性能指数，如前定义二所述，反映了节点的性能，调度时，需要按各节点的承受能力，即性能为其合理分配任务队列，另外，高复杂度的任务也应尽可能交由高性能节点执行，以减少低性能节点负载从而提高整体作业的效率。

Spark 上调度结构如图 1。

Driver 启动 Job 后，将会创建对应的 Task Scheduler 作为整个作业的任务调度器，当需要任务分配时，会根据任务所需资源向 Master 的集群管理器(Cluster Manager)申请运行资源，此时，Cluster Manager 会获取所以 Worker 节点上的可用 Executor 信息作为其任务的可分配资源池。当任务分配到 Executor 后，Executor 会向 Driver 注册并通过 Heartbeat 消息与其保持连接。此时可通过创建新的 Heartbeat 消息，将 CPU 状态同步传输给 Task Scheduler, Task Scheduler 则将其记录并保存，作为其调度指标。

Heartbeat 消息交互模型如图 2。

具体的交互流程如下：

1. Executor 被 Master 分配给 Driver 执行 Job 时，会向 Driver 发送注册信息，Driver 即获取了 Executor 的基本信息；

2. Executor 开始实例化并初始化执行时，会启动一个 Heartbeater，如图 2 所示，同时 Heartbeater 上回绑定一个 Heartbeat Task，即每次心跳时所执行的任务；

3. 每次心跳执行到该任务时，Executor 会创建一个 Heartbeat 消息，该消息包装了所需要向 Driver 传输的数据，可通过重构消息的结构完成自定义消息的传输；

4. Executor 发送消息后，会由 Driver 的 Heartbeat Receiver 接收到，并进行消息的解析，将上述的参数保存在 Driver 中，用于后续调度使用。

### 3.4. 集群分层调度方案

前三节已经分别从任务、集群和节点三个层次对 Spark 调度分别进行了分析。本文将综合上述三个因素，实现最终的调度策略。

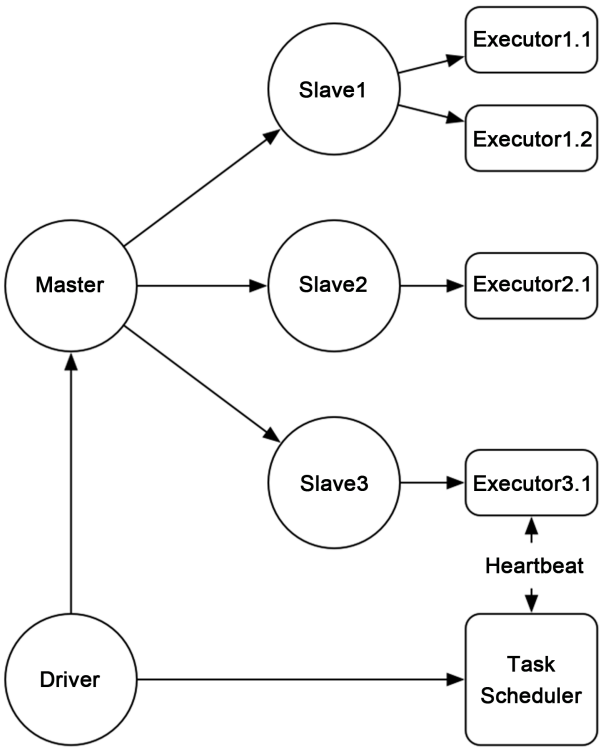


**Figure 1.** Spark cluster scheduling structure
**图 1.** Spark 节点调度结构

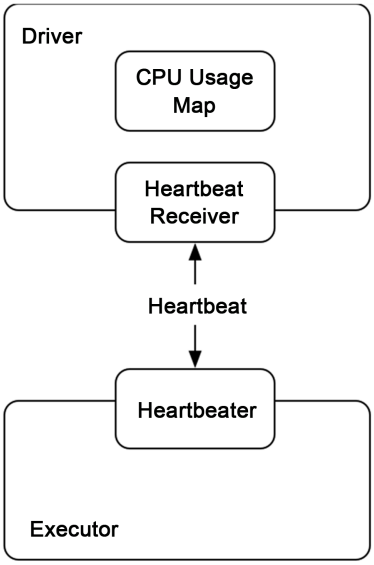


**Figure 2.** Heartbeat message interaction
**图 2.** Heartbeat 消息交互

首先，在任务执行前，将预先执行分层算法(算法一)，对当前的集群环境进行分析，并依据分析结果，将所有节点的性能指标数据写入 Spark 的配置文件，Spark 将会在初始化时读取并存入 SparkEnv 作为后续调度指标之一。该预执行作业会在以下情况发生时触发：

1. 集群硬件环境发生改变；
2. 分层数配置变化，如：从 3 层到 5 层；
3. 无层级信息数据；
4. 配置文件距上次更新时间过长，数据过期。

然后，Spark 开始正式启动，到创建 RDD 时，会解析传入的闭包操作复杂度，将其复杂度$C_p$记录到 RDD 上。Job 经过 DAG Scheduler 和 Task Scheduler 的划分，开始进行最终资源分配时，将会存在以下调度参数：对于 Task，将存在 $C_p$；对于待分配的 Executor，存在三元组($P_i$, *running Tasks.len*, *CPU Usage*)，分别代表该 Executor 的节点性能参数，其上执行队列的长度，以及 Task Scheduler 中记录的节点 CPU 使用情况。算法将在 Task Scheduler 中引入计算模块，计算每个 Executor 相对于待调度任务的得分，再结合其复杂度，完成 Executor 调度。

调度器系统结构如图 3。

**定义四：(评估函数)**定义 $len_q$ 为 Executor 上执行队列长度，$U_q$ 为其 CPU 使用率。定义 $f\left(len_q, U_q\right)$ 为 Executor 的当前执行能力指数。

$$f\left(len_q, U_q\right) = \theta \times len_q + \mu \times U_q \tag{2}$$

其中，$\theta$、$\mu$ 表示对应的评估指数，$len_{th}$ 表示执行队列长度阈值，$U_{th}$ 表示使用率阈值，当 Executor 的两个指标均超过阈值时，会降低该 Executor 的分配优先级，主要是防止出现单点密集分配的情况。函数值越小，说明执行能力越强。

**定义五：(复杂度转化函数)**定义 $g\left(C_p\right)$ 为复杂度转化函数，将复杂度转化为最优节点计算性能指数 $P_i$。调度时将会优先分配性能指数大于等于 $P_i$ 的节点。若未分配成功，则分配层级从 $P_i$ 向下依次选取，直至找到合适的 Executor。

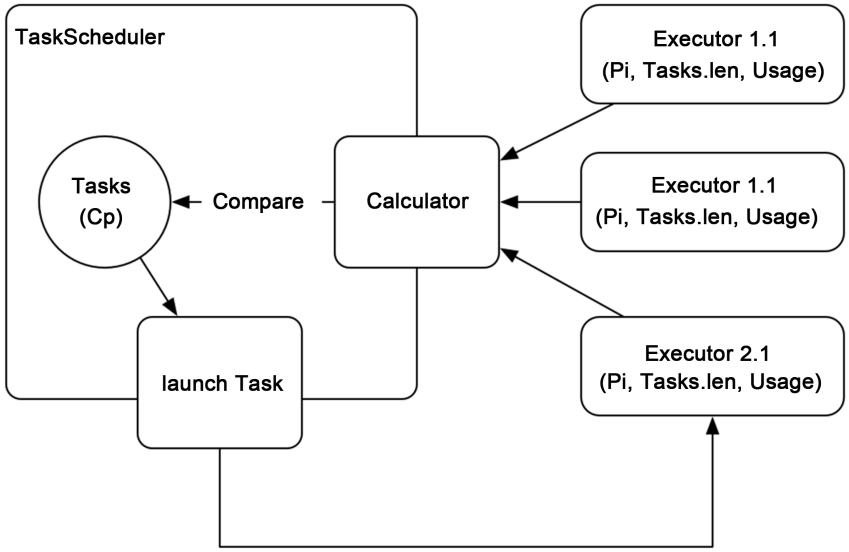


**Figure 3.** Scheduling strategy structure
**图 3.** 调度策略结构

$$g\left(C_p\right)=\frac{C_p}{C_{\max}}\times K \tag{3}$$

其中 $C_{\max}$ 表示所定义的性能指标的峰值。

**算法二 (节点调度算法)**

```
G[]                    //可执行 Executor 信息
selected = NULL        //所选 Executor
foreach Executor q
    if (lenq>lenth || Uq>Uth)
        break
    G[].add({q, f(lenq, Uq)})
end
G[]按升序排序
for (i = g(Cp) to K)
    for (j = 1 to G.size)
        level = get Executor G[j].q level
        if (level ≥i)
            selected = Executor G[j].q
    end
end
for (i = g(Cp)-1 to 1)
    for (j = 1 to G.size)
        level = get Executor G[j].q level
        if (level ≥i)
            selected = Executor G[j].q
```

```
        end
    end
    if (selected == NULL)
        selected = Random(Executors)
    return selected
```

算法的基本思路如下：

1. 首先遍历所有 Executor，排除所有队列长度或 CPU 使用率超过阈值的 Executor，并根据定义四的评估函数，得到当前每个 Executor 的可执行能力，并将队列升序排序 G；
2. 根据定义五的转化函数，计算得到当前复杂度的任务的最优执行层级；
3. 先遍历 G，得到大于等于最优执行层级的 Executor；
4. 若上一轮遍历没有满足的 Executor，则再次遍历，获取小于最优层级的 Executor；
5. 若最终依然没有结果，则随机调度。

## 4. 实验分析

实验部分将从算法仿真和真机两个角度进行调度策略分析，分析的指标主要是运行时间长短和各节点分配是否均衡。

### 4.1. 仿真实验

为验证调度策略本身的算法性能，本文通过代码仿真了异构集群环境下任务的生成和调度。实验思路如下：

1. 节点仿真：主要参数有性能指标、CPU 使用率、任务队列。节点参数根据正态分布生成。
2. 任务仿真：主要参数有执行时间、消耗 CPU。任务参数根据正态分布生成。任务的生成过程采用泊松分布。
3. 任务运行：任务被分配到节点运行时，消耗 CPU 将累计到节点 CPU 使用率上，并加入其任务运行队列。另外任务的执行时间将根据节点性能变化，即性能越高，所需要的执行时间越短。
4. 调度仿真：Spark 的原生策略用随机调度仿真，本文调度策略同上文所述。

本次仿真实验将多次统计作业的完成时间，以及集群各节点的使用率。运行时间的均值作为执行效率的指标，使用率作为分配是否均匀的指标。

**实验一：(相同任务数运行时间对比)**实验结果如图 4 所示。可见运行时间在相同执行节点条件和任务数下，新策略运行时间明显缩短。

**实验二：(相同任务数节点分配对比)** 实验结果如图 5 所示。可见在相同节点条件和任务数下，新策略的节点分布更加均匀。

**实验三：(不同任务数运行时间对比)** 实验结果如图 6 所示。可见相同节点条件下，随着任务数的增加，新策略的运行时间均短于原策略，且优势趋于明显(因为节点数少时，随机调度的大数原理相对不明显，导致调度过于集中而超过了单一节点的负载，时间明显变长)。

**实验四：(不同任务数节点分配对比)** 实验结果如图 7 所示。可见相同节点条件下，随着任务数的增加，节点的分配一直比原策略均匀，新策略在任务数增加依然可以保证均匀分配。

上述四个仿真实验，在模拟环境下推演了新策略的理论效果，且分别从运行时间和分配均匀度两个方面证明了新策略的优势，另外算法参数在实验中保持一致，实际使用中还可以通过多次实验对参数配置进行调优以提高执行效率，因此新策略是理论上是有效的。

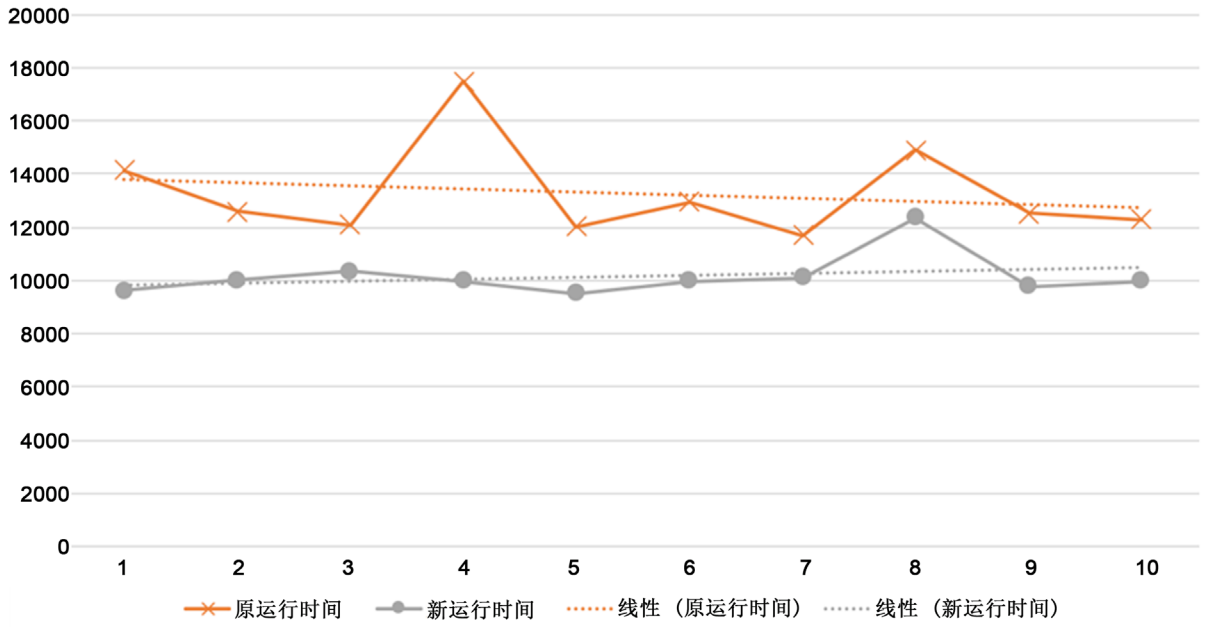


**Figure 4.** Time comparison of multiple experiment with the same task number
**图 4.** 相同任务数多次实验运行时间对比

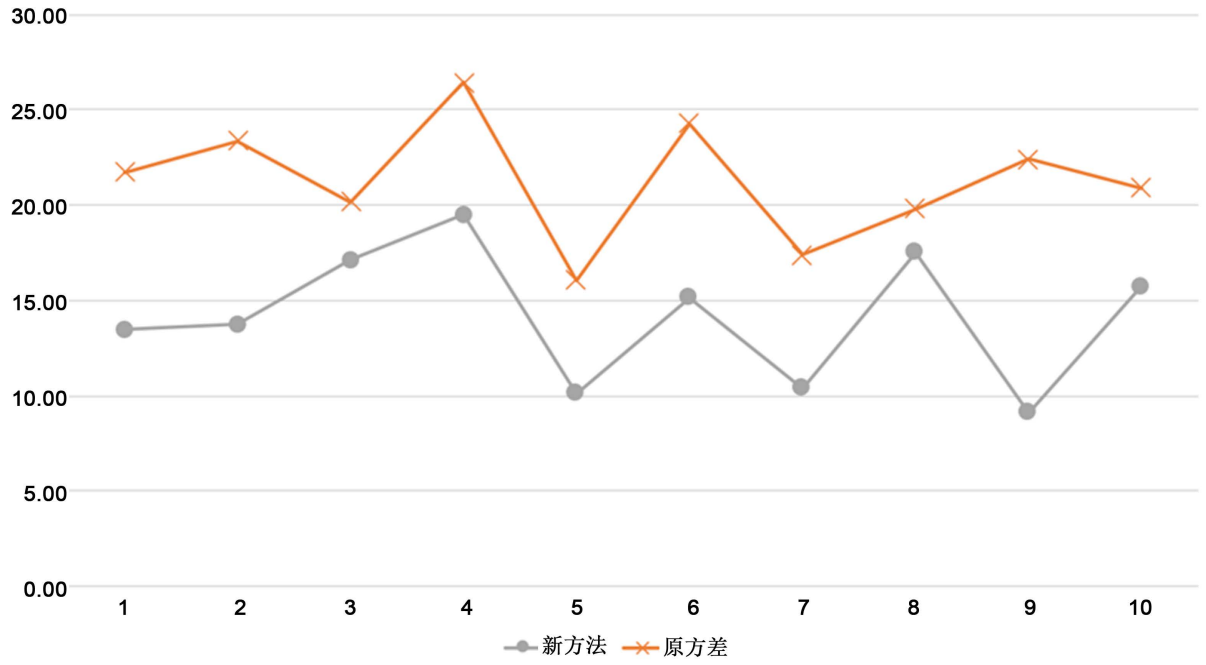


**Figure 5.** Distribution comparison of multiple experiment with the same task number
**图 5.** 相同任务数多次实验分配情况对比

## 4.2. 真机实验

上一节从仿真角度分析了新算法的优势。本节将于真机环境进行模拟实验。由于条件限制，无法完全模拟真实的异构集群环境，这里通过配置文件硬编码的方式，构建了虚拟异构集群。其中集群部署于阿里云，共 3 台机器，1 个 Master 加 3 个 Worker(Master 本身也作为 Worker)。配置为：Master(双核，4G

内存)，Worker(单核，2G 内存)。实验程序采用简单的 WordCount，然后在集群中多次运行不同大小的作业文件作为变量。

**实验五：(不同作业大小 WordCount 运行时间对比)** 实验结果如图 8 所示。图中运行时间为多次运行后的平均值，可见真实环境下，新策略确实缩短了 WordCount 的运行时间。

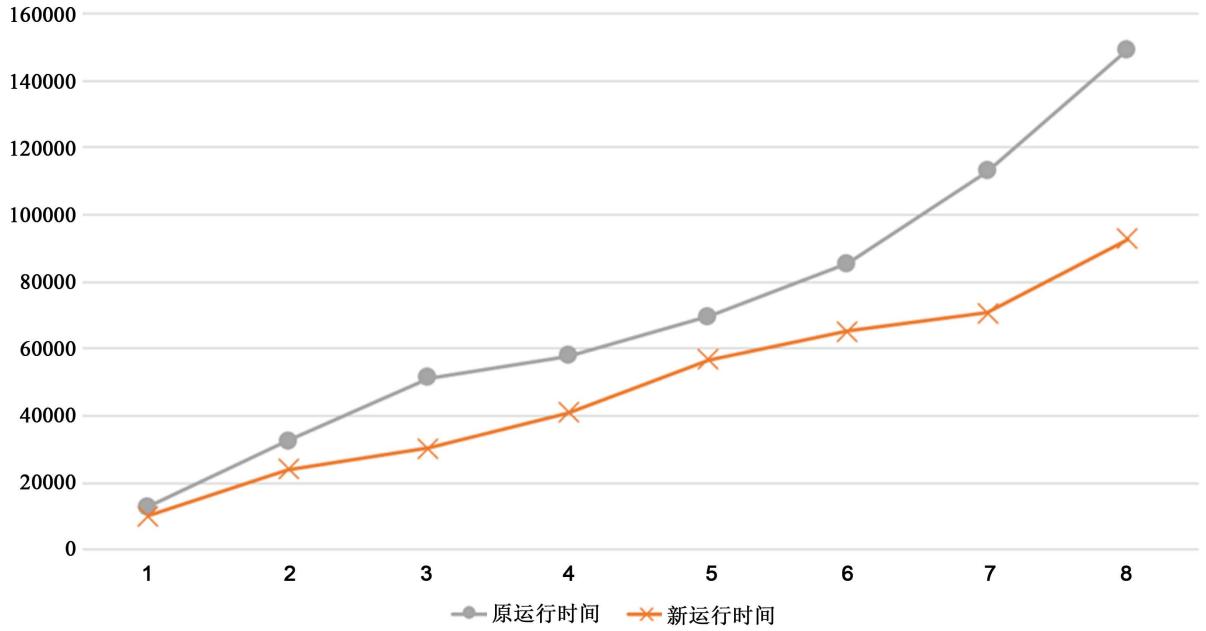


**Figure 6.** Time comparison of multiple experiment with different task number
**图 6.** 不同任务数多次实验运行时间对比

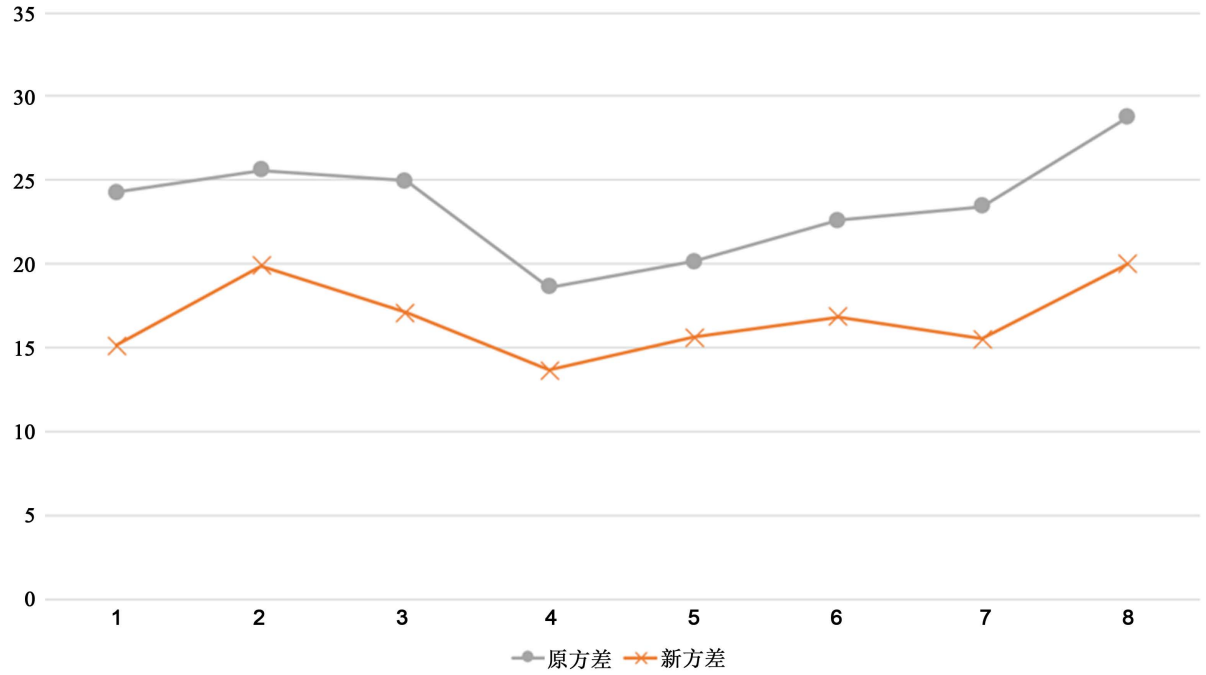


**Figure 7.** Distribution comparison of multiple experiment with different task number
**图 7.** 不同任务数多次实验分配情况对比

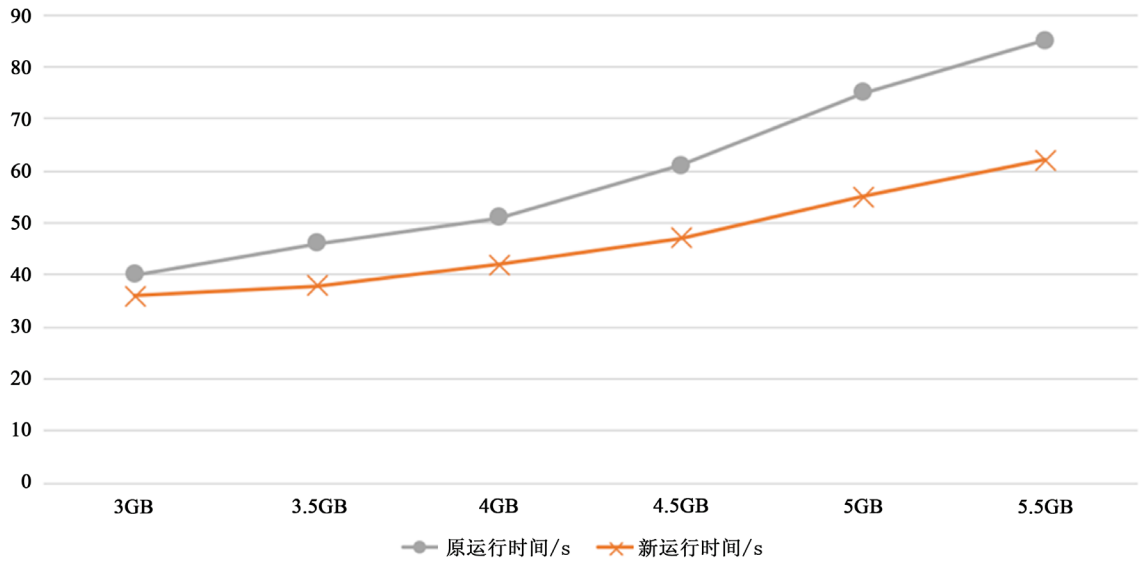


**Figure 8.** Time consumptions of multiple experiments with different job size
**图 8.** 不同作业大小多次实验运行时间对比

不过由于条件限制，实际的效果受到了多方面因素的影响，如实验环境并不是真实的异构集群，实验集群相对较小等。

但通过上述的仿真加真机模拟实验，仍可证明新策略相对于旧策略的性能优势。同时，还提供了新的提高集群执行效率的思路，即利用集群异构性，分层调度，以发挥高性能节点的计算优势，避免短板效应导致的整体集群运行效率下滑。

## 5. 结论

综上所述，新的调度策略在异构集群环境下可以让任务调度更均匀，并且减少作业整体的运行时间，提高集群的整体运行效率。该策略将切实地优化 Spark 在实际集群环境中的运行效率，有较高的使用意义。另外也提供了一种快速提升集群工作效率的方式，若作业中存在高计算量任务时，可通过在集群中引入少量 GPU 等高性能元件，然后使用该策略，Spark 将会识别出其中的高计算量任务并优先分配给这些高性能元件，这样整体的运作效率不会因为任务的复杂度提高而下降，同时也充分地利用了新引入的高性能元件，具备一定的工程意义。

下一步的工作将集中在如何优化本文中的细节算法，实现任务复杂度的自动识别以及如何用机器学习的手段自动对算法参数进行调优，从而让调度策略更智能。

## 致　谢

在此感谢国家自然科学基金项目的支持，感谢研究生导师刘功申先生的指导和帮助，也要向所有文献作者、研究前辈们表示感谢。

## 基金项目

国家自然科学基金项目(编号：61472248)。

## 参考文献 (References)

[1] Dean, J. and Ghemawat, S. (2008) MapReduce: Simplified Data Processing on Large Clusters. *Communications of the*

*ACM*, **51**, 107-113. https:/doi.org/10.1145/1327452.1327492

[2] Borthakur, D. (2007) The Hadoop Distributed File System: Architecture and Design. *Hadoop Project Website*, **11**, 21.

[3] Zaharia, M., Chowdhury, M., Franklin, M.J., *et al*. (2010) Spark: Cluster Computing with Working Sets. *HotCloud*, **10**, 10.

[4] Zaharia, M., Chowdhury, M., Das, T., *et al*. (2012) Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *Proceedings of the* 9*th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, 2.

[5] 杨志伟, 郑烃, 王嵩, 等. 异构 Spark 集群下自适应任务调度策略[J]. 计算机工程, 2016, 42(1): 31-35, 40.

[6] Thakur, S., Singh, R. and Sharma, S. (2015) Dynamic Capacity Scheduling in Hadoop. *International Journal of Computer Applications*, **125**. https:/doi.org/10.5120/ijca2015906178

[7] Zaharia, M. (2009) Job Scheduling with the Fair and Capacity Schedulers. *Hadoop Summit*, 9.

[8] Zaharia, M., Borthakur, D., Sen Sarma, J., *et al*. (2010) Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. *Proceedings of the* 5*th European Conference on Computer Systems*, ACM, 265-278.

[9] Nightingale, E.B., Chen, P.M. and Flinn, J. (2005) Speculative Execution in a Distributed File System. *ACM SIGOPS Operating Systems Review*, ACM, **39**, 191-205.

[10] Zaharia, M., Konwinski, A., Joseph, A.D., *et al*. (2008) Improving MapReduce Performance in Heterogeneous Environments. *OSDI*, **8**, 7.

[11] Yong, M., Garegrat, N. and Mohan, S. (2009) Towards a Resource Aware Scheduler in Hadoop. *Proceeding of ICWS*, 102-109.

[12] Tang, Z., Zhou, J., Li, K., *et al*. (2013) A MapReduce Task Scheduling Algorithm for Deadline Constraints. *Cluster Computing*, **16**, 651-662. https:/doi.org/10.1007/s10586-012-0236-5

[13] Xu, X., Cao, L. and Wang, X. (2014) Adaptive Task Scheduling Strategy Based on Dynamic Workload Adjustment for Heterogeneous Hadoop Clusters.