【电子与信息科学／Electronics and Information Science】

并发式Spark消息分发器

何玉林[1,2]，林泽杰[1,2]，徐毓阳[1]，成英超[1]，黄哲学[1,2]

1）人工智能与数字经济广东省实验室（深圳），广东深圳 518107；2）深圳大学计算机与软件学院，广东深圳 518060

摘　要：在大数据计算框架Spark中，驱动器采用迭代式消息分发机制，会增加任务提交的时间开销，影响任务执行的启动时间，限制了任务执行的并发性，导致多个执行器处于空闲等待状态，造成计算资源的浪费．使用线程池调度策略，构建一种高效且轻量级的并发式Spark消息分发器．与迭代式Spark消息分发器不同，并发式消息分发器更加关注且更适合调度开销较大的细粒度任务作业，通过解析包含执行器重要信息的元数据，获取任务列表及各个任务对应的执行器标识，创建线程池并为每个任务启动异步计算，从而实现并发式任务分发，在保证系统稳定和任务顺利执行的前提下，最大程度地减少任务分发的时间开销．在虚拟机构建的仿真集群环境上，通过与迭代式消息分发器进行对比，证实了并发式消息分发器的良好效果．实验结果表明，在内存保持不变的前提下，并发式Spark消息分发器可减少约9%的任务执行时间，同时能提高约5%的中央处理器的利用率．并发式Spark消息分发器有效解决了迭代式消息分发机制针对细粒度任务分发的时间开销过大和计算资源浪费的问题．

关键词：并行处理；大数据计算；Spark通信机制；消息分发；细粒度任务；线程池调度

中图分类号：TP391.9　　**文献标志码：**A　　**DOI：**10.3724/SP.J.1249.2025.03317

Concurrent Spark message distributor

HE Yulin[1,2], LIN Zejie[1,2], XU Yuyang[1], CHENG Yingchao[1], and HUANG Zhexue[1,2]

1) Guangdong Laboratory of Artificial Intelligence and Digital Economy (Shenzhen), Shenzhen 518107, Guangdong Province, P. R. China

2) College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, Guangdong Province, P. R. China

Abstract: In the Spark big data computing framework, the driver employs an iterative message distributor mechanism that incurs considerable task submission overhead, delays task initiation, restricts execution concurrency, and causes idle waiting among executors—ultimately leading to inefficient utilization of computing resources. To address these issues, we propose an efficient and lightweight concurrent Spark message distributor based on a thread pool scheduling strategy. In contrast to Spark's original distributor, the proposed design is better suitable for scheduling fine-grained, high overhead tasks. It parses metadata containing key executor information to extract the task list and corresponding executor identifiers to each task, then initializes a thread pool to launch asynchronous computations for each task, thereby enabling true concurrent task distribution. This approach significantly reduces dispatch latency while ensuring system stability and reliable task execution. Experimental evaluations conducted in a virtualized cluster environment demonstrate the superiority of the proposed distributor over the original Spark

Received: 2024-09-05; **Accepted:** 2025-02-21; **Online** (CNKI)**:** 2025-04-02

Foundation: Natural Science Foundation of Guangdong Province (2023A1515011667); Science and Technology Major Project of Shenzhen (KJZD20230923114809020); Basic Research Foundation of Shenzhen (JCYJ20210324093609026); Guangdong Basic and Applied Basic Research Foundation (2023B1515120020)

Corresponding author: Associate professor XU Yuyang (xuyuyang@gml.ac.cn)

Citation: HE Yulin, LIN Zejie, XU Yuyang, et al. Concurrent Spark message distributor [J]. Journal of Shenzhen University Science and Engineering, 2025, 42(3): 317-325. (in Chinese)

mechanism. Results show that, with memory usage held constant, the concurrent distributor reduces task execution time by about 9% and increases central processing unit utilization by about 5%. The proposed concurrent Spark message distributor, effectively mitigates the high overhead and computational resource inefficiency associated with traditional message distribution methods in fine-grained task scenarios.

Key words: parallel processing; big data computing; Spark communication mechanism; message distribution; fine-grained tasks; thread pool scheduling

如何快速有效地挖掘海量数据潜藏的有价值信息是互联网企业当前亟待解决的难题. 研究人员已开发了多种大数据处理框架，如支持批处理的Hadoop[1]、支持流处理的Storm[2]和Flink[3]，支持深度学习的Caffe[4]和Tensorflow[5]，以及支持内存计算的Spark[6]. 其中，Spark是一个高效、易用、可跨平台的大数据计算引擎，采用内存计算的模式，有效避免了频繁的磁盘访问，且支持多种计算模式和语言. Spark的核心数据被抽象为弹性分布式数据集[7](resilient distributed dataset，RDD)，具有较高的容错性和不可变性.

近年来，为提升Spark作业的执行效率，相继出现了一些针对执行器的改进方案. 从执行器自身优化出发，KHORASANI等[8]提出自适应执行器，通过监视输入和输出的等待时间和吞吐量来动态调整线程数量，显著缩短了执行时间. BARESI等[9]设计了一个容器级控制器，利用反馈循环机制监视执行器的进度并调整资源分配. MORISAWA等[10]提出执行器预热机制，通过自动创建和管理预热作业减少初始化开销，并动态调整流应用程序. LI等[11-12]提出可对执行器的参数进行自动调优的自适应模型. FU等[13-14]提出最小化总体通信代价的最优任务调度算法和基于位置感知的执行器分配策略来对Map和Reduce阶段的数据局部性进行优化. SHABESTARI等[15]提出基于贪心算法、随机算法和帕累托法则的执行器选择方法. LI等[16]提出一种异构集群下使用机器学习进行资源需求预测的低成本执行器放置方法. TARIQ等[17]提出基于图的确定性分析模型，用于预测Spark应用程序的执行时间，从而决定是否分配或释放执行器. CHENG等[18]提供了一种基于感知的资源供应方法，通过监控执行者的资源使用情况灵活调整分配的执行器数量. SAHIN等[19]采用弹性执行器设计了一个轻量级的RDD优化器，通过内存管理器回收内存以优化资源分配. LIU等[20]提出一种新的调度器，可根据资源需求动态分配执行器以最大限度地减少资源碎片. KATSOGRIDAKIS等[21]采用本地任务队列策略构建多个代理调度器，用于优先处理本地任务. XU等[22]设计了一种用于并行任务调度的中间件，不断更新有向无环图中各阶段任务的优先级，并为每个阶段动态分配执行器资源，并做出任务分配决策. 上述策略可总结为：① 监控执行器状态调整对执行器的资源分配；② 优化执行器的分配和释放策略；③ 改进调度器用以优化分配任务给执行器的过程. 可见，执行器的性能对作业的执行十分重要. 然而，前两种并未对分发消息进行细粒度的优化，最后一种则忽略了对分发消息起重要作用的驱动器的优化.

本研究对驱动器向执行器的分发消息策略进行优化和完善，提出一种并发式Spark消息分发器. 该分发器能够在保证任务正确执行的前提下，加快任务的分发速度，实现更高效地使用集群资源，缩短作业的执行时间. 研究的主要贡献为：① 创新性地引入多线程和异步编程技术，对任务进行分析，将对应不同执行器的任务同时进行分发，以满足Spark环境下执行多样化任务的需求；② 在搭建的虚拟机环境中，使用多种算法和数据集评估新的并发式Spark消息分发器的性能，证实其可以降低任务的通信延迟，提高作业执行效率.

1 Spark通信原理

1.1 Spark基本架构

从集群部署的角度来看，Spark集群由应用程序(application)、集群管理器(cluster manager)、工作节点(worker node)、驱动器(driver)和执行器(executor)等部分组成. 集群管理器主要负责协调和管理应用程序在整个集群中的资源，包括资源分配、任务调度和容错处理等功能，目标是确保应用程序在集群中被稳定、有序且正确地执行. 工作节点是实际执行计算任务的节点，负责处理应用程序的任务. 驱动器用于实际代码的执行工作. 执行器负责具体任务的执行. 在执行Spark作业时，驱动器负责将应用程序转化为作业、分发任务给执行

器、监控执行器的执行情况和接收执行器返回的运行结果．用户提交的应用程序通过驱动器和资源管理器来与执行器通信．

1.2　YARN集群运行模式

Spark支持多种集群管理器，本研究使用Spark on YARN集群管理器．针对Spark，YARN有客户端(client)和集群(cluster)两种运行模式．二者的区别在于驱动器运行位置不同，前者运行于客户端，而后者运行于工作节点．集群模式下的通信效率高、稳定性强，不受客户端进程影响，下面将从用户提交执行命令开始，描述Spark的YARN cluster模式的运行方式．

首先，用户使用spark-submit命令行指定集群模式和执行脚本并提交任务．然后，Spark的客户端与YARN的资源管理器(resource manager，RM)通信．RM选择在合适的节点管理器(node manager，NM)上启动应用程序主控(application master，AM)．AM启动驱动器线程，用于执行用户的作业．然后，AM向RM注册，申请执行器的资源．RM采用轮询的方式为各任务申请容器资源，分配给NM并在其上启动执行器进程，同时监控NM的运行状态直到运行结束．各节点的执行器会向驱动器反向注册．全部执行器反向注册完成后，执行器启动计算对象并等待接收任务．

1.3　驱动器与执行器之间的通信

Spark的通信架构涉及4种消息传递，以YARN作为Spark的资源管理器为例，分别为驱动器与资源管理器RM之间的通信、RM与NM之间的通信、驱动器与执行器之间的通信，以及执行器(或NM)之间的通信．其中，驱动器与执行器之间的通信体现在执行器向驱动器申请注册计算对象，以及申请成功后驱动器向执行器分发任务的过程．执行器需向AM中的驱动器线程进行注册，让驱动器记录可分配任务的执行器信息．所以，一旦执行器被RM成功创建，会立刻向驱动器发送RegisterExecutor消息，请求在驱动器内部注册执行器信息．执行器收到注册成功的消息后，会在内部实例化一个负责计算的对象executor．执行器的驱动器申请注册计算对象成功后，驱动器统计所有已注册的执行器，并对调度器发送的任务集TaskSet进行处理．驱动器通过TaskSet与执行器之间的对应关系，迭代式地向执行器发送任务．执行器成功接收任务后，对任务进行分解，并使用内部的executor对象执行任务．

1.4　RPCEndpoint通信原理

Spark的通信部分包括很多组件，如与任务执行相关的驱动器和执行器，它们都继承了远程过程调用(remote procedure call，RPC)端点RPCEndpoint．可以说，RPCEndpoint代表终端，而各组件是独立的实体，所有实体都继承了此RPCEndpoint．一个终端可在通信环境中与其他终端进行通信．RPC环境(RPC environment，RPCEnv)是各组件之间通信的执行环境，集群中每个节点的RPCEndpoint之间的通信都通过RPCEnv协调，底层则使用Netty通信框架．

RPCEndpoint主要负责处理消息，而RPCEndpointRef是RPCEndpoint对应的引用．派发器作为中间件对消息去向进行管理，它将RPCEndpointRef发送的消息先存入消息发件箱(OutBox)，再通过通信客户端(TransportClient)发送给对应的RPC端点．通信服务端(TransportServer)负责接收其他RPC端点发送的消息，然后调用派发器将消息分发到消息收件箱(Inbox)，RPCEndpoint会从Inbox中获取消息并处理．驱动器和执行器都继承了RPCEndpoint，它们的信息发送和接收都是基于上述RPCEndpoint的通信方式进行的．

2　并发式Spark消息分发器

2.1　框架描述

并发式Spark消息分发器的目标是当集群进行多任务处理时，能够在保证任务正确执行的前提下，加快任务的分发速度，缩短了执行器的等待时间，实现更高效地使用集群资源，缩短了作业的执行时间．

图1为本研究提出的并发式消息分发器的整体架构．其中，驱动器中任务列表(tasks)的每个任务(TaskDescription)都保存了目标执行器的唯一标识，消息分发器接收含有不同标识的任务后，并发式地向多个目标执行器发送消息．

2.2　关键步骤及实现原理

并发式Spark消息分发器的实现需要执行器向驱动器注册、驱动器并发式分发任务给执行器和执行器对任务进行处理3个步骤．

1）执行器向驱动器注册．首先，在Spark执行环境中创建Shuffle客户端传输配置，若其启用了Netty的直接内存，且最大直接内存的大小应小于

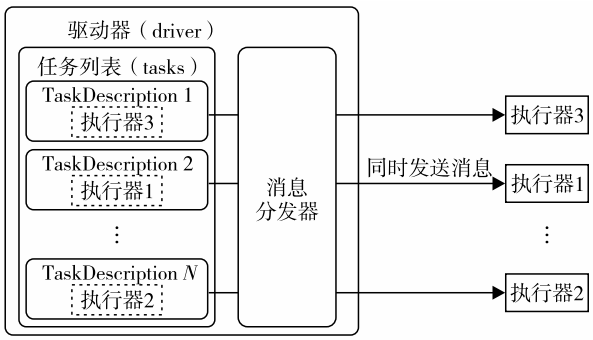


图1 并发式消息分发器模型

Fig. 1 Concurrent message dispatcher model.

配置的值，则抛出异常．然后，解析资源配置文件并将结果存储于_resources．接着，通过驱动器的地址（driverURL），在执行器内部注册驱动器（driver），并向其发送包含执行器信息的RegisterExecutor消息．若执行器得到驱动器的确认回复，则向执行器发送RegisterExecutor消息，创建用于后续执行任务的计算对象executor．执行器向驱动器注册的伪代码请扫描论文末页右下角二维码查看补充材料图S1．

驱动器收到执行器的注册请求后进行回复．驱动器的ExecutorDataMap属性会保存已注册的执行器信息．驱动器收到RegisteredExecutor消息后，根据ExecutorDataMap属性判断发送RegisteredExecutor消息的执行器是否已在驱动器中注册．若未注册，则创建ExecutorData对象，并将其保存到包含执行器信息的键值对ExecutorDataMap中；若已注册，则驱动器向执行器返回注册成功（success）消息，确保两者之间能够正常通信．驱动器接收执行器发送的RegisteredExecutor消息的伪代码请扫描论文末页右下角二维码查看补充材料图S2．

执行器请求驱动器发布任务．执行器接收到驱动器发送的success消息后，根据执行器身份证标识号（identity document，ID）、因特网协议（Internet protocol，IP）地址和资源信息_resources，创建执行器中的计算对象executor，用于后续任务执行．然后，执行器向驱动器发送LaunchExecutor消息，请求驱动器向该执行器发布任务．LaunchExecutor消息包含此执行器的信息，用于驱动器确定发送方的身份及位置．

2）驱动器并发式分发任务给执行器．首先，在驱动器内部创建执行器与互斥锁之间的映射ExecutorLocks，目的是保存被分配到任务的执行器对应的锁，用于后续分发任务的线程安全．其中，ExecutorLocks使用编程语言Scala中的map结构；互斥锁使用Java中的ReentrantLock．然后，根据包含执行器重要信息的元数据，使用调度器的resourceOffers方法解析ExecutorData，从而获取待执行的任务列表taskDescs．驱动器利用Scala的异步编程特性，创建1个隐式的执行上下文ExecutionContext，并使用1个固定大小的线程池，将任务列表taskDescs展平为一维序列，同时为每个任务创建1个异步计算的结果Future．其中，ExecutionContext允许在Scala中进行异步编程，而每个任务可通过Future启动异步计算，在新线程执行分发任务函数LaunchTask．在分发任务函数中，先根据执行器ID，对相应的执行器上锁，保证对同一执行器的访问是线程安全的，再将任务序列化并封装成LaunchTask消息，然后发送给对应的执行器．所有任务分发结束后，关闭线程池，等待各执行器返回结果．通过以上方法将逐个向执行器发送任务的机制变为同一时间向多个执行器分发任务．驱动器并发式分发任务给执行器的伪代码请扫描论文末页右下角二维码查看补充材料图S3．

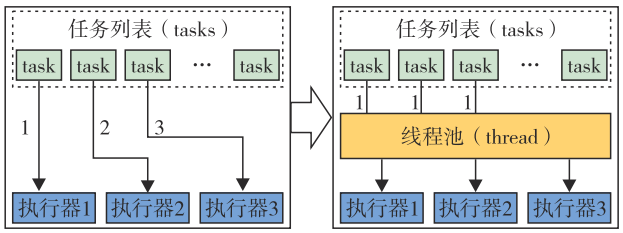
3）执行器对任务进行处理．执行器接收驱动器发送的LaunchTask消息后对任务进行处理．执行器先将序列化后的任务serializedTask反序列化，再创建1个运行该任务的TaskRunner对象，并将该对象添加到正在运行的任务集合runningTasks中．将TaskRunner对象放入线程池中，以启动任务的执行．最后，执行器创建StatusUpdate对象，写入执行结果并发送到驱动器汇总．其中，任务是异步执行的，由线程池来管理．

2.3 优化效果

迭代式消息分发器和并发式消息分发器的运行流程对比如图2．迭代式消息分发器将任务逐个发送给对应的执行器，而并发式消息分发器采用多线程并发的形式，将多个任务同时发送给对应的执行器，因此提高了消息分发的效率．

对于有依赖性的作业，迭代式Spark消息分发器会根据任务之间的依赖关系来确定任务的调度和执行顺序，保证任务执行过程中能正确处理任务失败和数据丢失等异常情况，确保作业能够正确执行．

然而，迭代式消息分发器并未对细粒度任务进行性能优化．通常情况下，Spark上作业时的细粒度任务数量较多，调度的通信开销占据了整个任务



（a）迭代式消息分发器　　（b）并发式消息分发器

图2　迭代式消息发发器和并发式消息分发器的运行流程对比

Fig. 2　Comparison of running processes of (a) iterative message distributor and (b) concurrent message distributor.

执行的大部分时间，若分发任务的速度不够快，可能会成为系统运行的瓶颈．并发式消息分发器利用细粒度任务之间的独立性，使用线程池调度策略分发细粒度任务，令多个执行器能够同时接收任务，有效减少了任务分发的时间开销，从而提升了系统的整体性能．

并发式消息发送的效果依赖于任务量的规模和驱动器与执行器间的通信开销．基于分布式计算中任务管理的实践经验和任务量对系统性能的影响规律，可将任务量划分为小任务量（$<1\times10^3$ 个）、中等任务量（$1\times10^3\sim1\times10^4$ 个）和大任务量（$>1\times10^4$ 个）．针对3种任务量，分析两种消息发送的效果：

1）针对小任务量，迭代分发任务的时间开销较低，采用并发方式并不能显著提升性能，甚至在计入消息初始化和线程管理等额外开销后，时间开销可能超过迭代分发；

2）针对中等任务量，迭代分发的通信瓶颈开始显现，并发分发能够显著降低驱动器的单线程压力；

3）针对大任务量，并发方式通过分摊任务分发的开销，提高了吞吐量，比迭代分发的优势更显．

任务数量通常与任务粒度直接相关．对于大粒度任务（如批处理场景），1×10^3 个任务可能已足够覆盖整个计算任务；细粒度任务（如迭代计算或大规模数据分析）的任务数量容易达到数万，如后续测试使用的k均值（k-means）聚类算法和网页排名（PageRank）算法，任务的数量通常处于$10^3\sim10^5$数量级．

设总任务数为N，分发每个任务的时间开销为t_s（迭代分发）或t_c（并发分发），驱动器的并发线程数为P，系统的其他固定时间开销为t_{other}，则总时间开销t_{seq}（迭代分发）或t_{conc}（并发分发）分别为

$$t_{seq}=Nt_s+t_{other} \tag{1}$$

$$t_{conc}=(N/P)t_c+t_{other} \tag{2}$$

由式（1）和式（2）可见，N越大，迭代分发的瓶颈愈明显，并发优势愈强．当$t_c\ll t_s$时，并发收益更显著．设$N=1\times10^4$ 个，则$t_s=10$ ms，$t_c=2$ ms，$t_{other}=2\,000$ ms，若$P=50$，则$t_{seq}=1.02\times10^5$ ms，$t_{conc}=2\,400$ ms，可见并发性能具有显著优势．

3　性能评估

3.1　实验设置

使用VMware Workstation 17Pro虚拟机[23]构建Spark集群仿真环境．集群由3个节点组成，每个节点使用内存大小为9 Gbyte的6核中央处理器（central processing unit，CPU）．Hadoop版本为3.3.0，Spark版本为3.2.4．参考文献[24]的设置，使用Ganglia（3.7.2版）实时监控集群中CPU和内存等资源的使用情况．

分别使用k-means算法和PageRank算法对并发式消息分发器与迭代式消息分发器的性能进行实验验证．二者都将数据分解为多个小数据，每个小数据都是一个独立的子任务，适合作为细粒度任务进行测试．使用Spark示例代码对两种分发器进行测试，k-means算法和PageRank算法分别取自MLlib和GraphX模块．k-means算法使用如表1的前5个高斯聚类合成数据集，每个数据集都有100个特征和25个中心点．PageRank算法使用表1中的后4个生成的Kronecker图[25]数据集．根据数据集的大小，Hadoop分布式文件系统（Hadoop distributed file system，HDFS）的块大小设置为64 Mbyte．

表1　实验所用数据集信息

Table 1　The information of datasets

数据集	$10^{-5}\times$样本点数量/个	维度	文件大小/Mbit
KM-500K	5	100	354
KM-1M	10	100	708
KM-1200K	12	100	851
KM-1400K	14	100	992
KM-1600K	16	100	1 135
PR-1500K	15	9 635 820	142
PR-2M	20	11 361 025	172
PR-2500K	25	14 106 752	216
PR-3M	30	18 106 752	280

3.2 实验结果

从合理性和有效性两方面对并发式消息分发器与迭代式消息分发器在实际应用中的性能进行分析验证.

3.2.1 合理性

对于k-means算法，使用KM-500K、KM-1M、KM-1200K、KM-1400K和KM-1600K数据集考察原Spark框架(迭代式消息分发)和改进后的Spark框架(并发式消息分发)的误差平方和(sum of squares due to error, SSE)和来进行合理性验证. 其中，SSE计算的是拟合数据和原始数据对应点的SSE. 改进前后Spark框架对5个数据集测试后的SSE差值不超过0.1×10^8(实验结果请扫描论文末页右下角二维码查看补充材料表S1)，说明两模型设计具有合理性.

使用PR-1500K、PR-2M、PR-2500K和PR-3M数据集，对比Spark输出的前10个最重要页面及其对应的PageRank值(结果请扫描论文末页右下角二维码查看补充材料表S2)，以进行合理性验证. 实验结果发现，改进前后输出的前10个最重要页面完全一致，说明并发式消息分发器不影响PageRank算法的运行结果，模型设计合理.

3.2.2 有效性

根据文献[26]，使用加速比(speedup)、扩展比(scaleup)、伸缩率(sizeup)和资源利用率等指标对不同消息分发器的性能进行评估. 其中，加速比是同一数据集在不同规模集群上运行相同算法的时间之比；扩展比是不同大小的数据集在不同规模集群上运行相同算法的时间之比，数据集的大小和集群的规模成正比；伸缩率是不同规模数据集在同一集群上运行的时间之比. 前3个指标直接反映了优化后的系统在实际任务执行中的整体性能，资源利用率指标通过CPU利用率和内存占用大小全面评估系统资源的利用效率，以及并发处理机制对系统性能的具体影响，检验多线程的实际运行效率、识别是否存在过载情况. 加速比、扩展比和伸缩率中的执行时间由Spark用户界面(user interface, UI)提供，资源利用率由Ganglia监控系统提供.

使用相同数据集、不同数量的执行器，比较k-means算法和PageRank算法进行大规模数据处理时的执行时间，并分析对应的加速比指标，结果如表2. 其中，每个执行器的CPU个数为1，用于执行k-means算法的执行器内存为1 Gbyte，用于执行PageRank算法的内存为2 Gbyte. 由表2可见，对于k-means算法和PageRank算法，使用相同的数据集，随着执行器数量增加，加速比保持稳定，说明并发式分发器不受资源的影响，只针对多个细粒度任务进行了优化.

表2 加速比指标测试结果

Table 2 Speedup testing results

数据集	执行器数量/个	执行时间/s		加速比
		迭代分发	并发分发	
KM-1200K	3	759	681	1.115
	6	439	396	1.109
	9	492	441	1.116
PR-2500K	3	351	317	1.107
	4	326	292	1.116

表3为扩展比指标的结果，即按照比例增加数据集和执行器数量，比较k-means算法和PageRank算法的执行时间. 其中，k-means算法的数据集按照1∶2∶3的比例设置；PageRank算法按照3∶4∶5的比例设定；每个执行器参数与上述实验设置一致. 由表3可见，对于k-means算法和PageRank算法，使用相同比例的数据集和资源时，扩展比稳定大于1，说明改进后的Spark框架具有一定的扩展性，使其能够适用大规模数据集在集群环境下的处理和分析.

表3 扩展比指标测试结果

Table 3 Scaleup testing results

数据集	执行器数量/个	执行时间/s		扩展比
		迭代分发	并发分发	
KM-500K	3	216	193	1.119
KM-1M	6	422	375	1.125
KM-1200K	9	492	441	1.116
PR-1500K	3	211	179	1.179
PR-2M	4	254	236	1.076
PR-2500K	5	390	371	1.051

表4为伸缩率指标的结果，即在相同配置下，使用不同数据集，比较k-means算法和PageRank算法的执行时间. 其中，每个执行器的CPU个数为2，内存为2 Gbyte. 由表4可见，对于k-means算法和PageRank算法，使用相同配置，随着数据集越大，执行时间缩短幅度更大. 比如，前者在使用KM-1200K数据集的情况下，伸缩率高达1.189，执行时间缩短了15%，后者在使用PR-2500K数据集的情况下，伸缩率高达1.104，执行时间缩短了

表4　伸缩率指标的测试结果

Table 4　Sizeup testing results

数据集	执行器数量/个	执行时间/s		伸缩率
		迭代分发	并发分发	
KM-500K	3	312	304	1.026
KM-1M	3	466	421	1.107
KM-1200K	3	661	556	1.189
PR-1500K	3	301	291	1.034
PR-2M	3	342	314	1.089
PR-2500K	3	437	396	1.104

9%，说明并发式分发器在处理更大规模数据时，细粒度任务数量越多，性能改进越明显.

值得注意的是，并发式消息分发器性能的提升并不是以大幅提升资源开销为代价的. 以伸缩率指标为例(表4)，在PR-2500K数据集上改进前的平均CPU利用率约为38.4%，平均内存占用大小约为6.94 Gbyte，作业执行时间为437 s. 改进后的平均CPU利用率约为40.8%，平均内存占用大小为约为7.56 Gbyte，作业执行时间为396 s. 设改进前后的平均CPU利用率分别为U_1和U_2，改进前后的平均内存占用大小为M_1和M_2，改进前后的执行时间分别为t_1和t_2，CPU成本为C_1，内存成本为C_2，则改进前后总的开销变化为

$$\Delta E = \text{改进前的开销} - \text{改进后的开销} = C_1(U_1t_1 - U_2t_2) + C_2(M_1t_1 - M_2t_2) > 0 \quad (3)$$

由式(3)可见，并发式分发器在性能提升的同时，作业执行的资源开销实际上是在降低的.

表5为*k*-means算法运行的任务调度时间和任务执行时间. 其中，任务调度时间为执行时间最长的阶段内多个任务调度时间的中位数；执行时间为第1个任务结束到整个作业完全结束的时间. 由表5可见，对于*k*-means算法，使用相同配置，随着数据集越大，调度时间和执行时间减少的幅度更大，优化效果更加明显.

表5　*k*-means算法的任务调度与执行时间

Table 5　Task scheduling and execution time of *k*-means algorithm

数据集	调度时间/s		执行时间/s	
	迭代分发	并发分发	迭代分发	并发分发
KM-1M	8	5	326	309
KM-1200K	18	13	450	419
KM-1400K	22	15	570	514
KM-1600K	28	18	656	580

资源利用率包括CPU利用率和内存占用大小. 图3是*k*-means算法在数据集KM-1M和KM-1200K上的运行结果. 这两个数据集的数据量较大，能够更好地验证并发式消息分发器的性能，PageRank算法的数据集PR-2500K和PR-3M选择同理. 由图3可见，*k*-means算法运行过程中出现了2个资源使用峰值，第1个高峰处于分配任务的阶段，第2个高峰处于最后计算结果汇总的阶段. 随着程序的执行，改进后的Spark比原Spark的CPU利用率高，KM-1M数据集上的CPU利用率最高达17%；KM-1200数据集的CPU利用率最高达到23%. 对于内存的占用大小与原先的几乎保持一致.

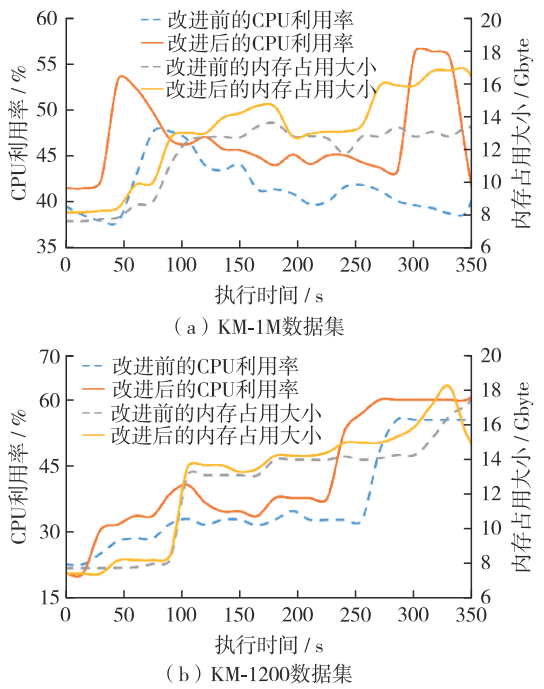


图3　KM-1M数据集和KM-1200数据集上*k*-means算法的资源利用率

Fig. 3　Resource utilization of *k*-means algorithm on (a) KM-1M dataset and (b) KM-1200 dataset.

图4为PageRank算法在数据集PR-2500K和PR-3M上的运行结果. 由图4可见，PageRank算法在作业运行过程中，资源使用率先升后降，原因是PageRank算法的每一轮迭代都需要计算节点的排名，导致资源利用率(如CPU利用率)升高. 然而，随着迭代的进行，算法逐渐收敛，每次迭代所需的计算量逐渐减少，因此，CPU利用率逐渐降低. 随着程序的执行，改进后的Spark比原Spark的CPU利用率高，对于PR-2500K数据集，最高达到18%；对于PR-3M数据集，最高达到20%. 对于内存的占

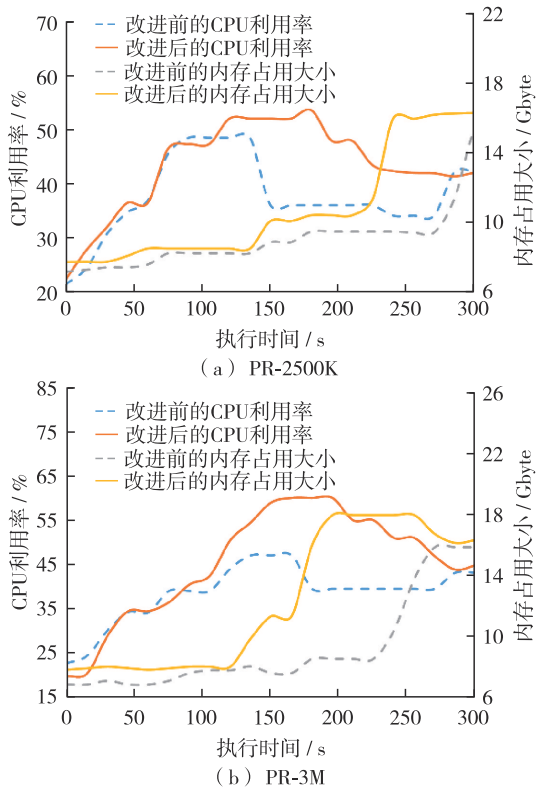


图 4 PageRank 的资源利用率

Fig. 4 Resource utilization of PageRank on (a) PR-2500K dataset and (b) PR-3M dataset.

用大小，在短时间内可能会比原先高出很多，是因为消息分发的速度提升，使得迭代提前执行，而二者峰值之间不超过 2 Gbyte，并不会造成内存溢出.

4 结 论

提出一种并发式 Spark 消息分发器，能够在保证任务正确执行的前提下，加快任务的分发速度，更高效地使用集群资源，缩短作业的执行时间. 在由虚拟机搭建的仿真集群环境上，通过与原始的消息分发器进行对比，验证了改进后的 Spark 消息分发器的良好效果. 分别使用 k-means 和 PageRank 算法对生成的 9 个数据集进行合理性和有效性验证，分析加速比、扩展比和伸缩率指标. 实验结果表明，所提并发式消息分发器能够减少程序的执行时间，且在内存不变的前提下，提高 CPU 利用率. 所提并发式消息分发器的优势总结如下：

1）扩展性良好. 能根据现有资源规模和任务分布情况优化任务分配，使任务在扩展后的系统中更高效地执行，从而提高系统整体性能，有助于大数据在集群环境下的高效处理与分析.

2）可实现性强. 采用线程池调度策略，降低配置和资源管理的复杂性，所有参数由 Spark 自动设定，用户无需关心底层细节，只需关注业务逻辑和数据处理，提高了用户使用体验和开发效率.

为改善 Spark 消息处理机制的性能，未来研究将从以下两方面探索行之有效的解决方案：

1）对任务的接收端（即执行器）进行优化，改进心跳信息机制，实时监测任务的执行情况，及时发现心跳丢失和心跳频率调整问题，避免任务失败或执行缓慢对系统造成的影响.

2）采取更加高级的消息分发策略，如将多个小任务合并为一个批次处理、将高优先级的 CPU 密集型任务分配到较空闲的 CPU 资源节点上，进一步提高消息分发的效率和集群资源的利用率.

基金项目： 广东省自然科学基金资助项目（2023A1515011667）；深圳市科技重大专项资助项目（KJZD20230923114809020）；深圳市基础研究资助项目（JCYJ20210324093609026）；广东省基础与应用基础研究基金粤深联合基金重点资助项目（2023B1515120020）

作者简介： 何玉林(yulinhe@gml.ac.cn)，人工智能与数字经济广东省实验室（深圳）研究员. 研究方向：大数据智能处理与分析.

引　　文： 何玉林，林泽杰，徐毓阳，等. 并发式 Spark 消息分发器[J]. 深圳大学学报理工版，2025，42(3)：317-325.

参考文献 / References：

[1] VAVILAPALLI V K, MURTHY A C, DOUGLAS C, et al. Apache Hadoop YARN: yet another resource negotiator [C]// Proceedings of the 4th Annual Symposium on Cloud Computing. New York, USA: ACM, 2013: 1-16.

[2] IQBAL M H, SOOMRO T R. Big data analysis: Apache storm perspective [J]. International Journal of Computer Trends and Technology, 2015, 19(1): 9-14.

[3] KATSIFODIMOS A, SCHELTER S. Apache flink: stream analytics at scale [C]// International Conference on Cloud Engineering Workshop (IC2EW). Piscataway, USA: IEEE, 2016: 193-193.

[4] JIA Yangqing, SHELHAMER E, DONAHUE J, et al. Caffe: convolutional architecture for fast feature embedding [C]// Proceedings of the 22nd ACM International Conference on Multimedia. New York, USA: ACM, 2014: 675-678.

[5] ABADI M, BARHAM P, CHEN Jianmin, et al. TensorFlow: a system for large-scale machine learning [C]// Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. USA: USENIX Association, 2016: 265-283.

[6] ZAHARIA M, XIN R S, WENDELL P, et al. Apache

Spark: a unified engine for big data processing [J]. Communications of the ACM, 2016, 59(11): 56-65.

[7] ZAHARIA M, CHOWDHURY M, DAS T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing [C]// Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. San Jose, USA: USENIX Association, 2012: 2.

[8] KHORASANI S O, RELLERMEYER J S, EPEMA D. Self-adaptive executors for big data processing [C]// Proceedings of the 20th International Middleware Conference. New York, USA: ACM, 2019: 176-188.

[9] BARESI L, DENARO G, QUATTROCCHI G. Symbolic execution-driven extraction of the parallel execution plans of Spark applications [C]// Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, USA: ACM, 2019: 246-256.

[10] MORISAWA Y, SUZUKI M, KITAHARA T. Flexible executor allocation without latency increase for stream processing in Apache Spark [C]// International Conference on Big Data (Big Data). Piscataway, USA: IEEE, 2020: 2198-2206.

[11] LI Yang, JIANG Huaijun, SHEN Yu, et al. Towards general and efficient online tuning for Spark [J]. Proceedings of the VLDB Endowment, 2023, 16(12): 3570-3583.

[12] LIN Chen, ZHUANG Junqing, FENG Jiadong, et al. Adaptive code learning for Spark configuration tuning [C]// The 38th International Conference on Data Engineering. Piscataway, USA: IEEE, 2022: 1995-2007.

[13] FU Zhongming, TANG Zhuo, YANG Li, et al. An optimal locality-aware task scheduling algorithm based on bipartite graph modelling for Spark applications [J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(10): 2406-2420.

[14] FU Zhongming, HE Mengsi, TANG Zhuo, et al. Optimizing data locality by executor allocation in reduce stage for Spark framework [C]// Parallel and Distributed Computing, Applications and Technologies. Cham, Switzerland: Springer International Publishing, 2022: 349-357.

[15] SHABESTARI F, JAFARI NAVIMIPOUR N. An energy-aware resource management strategy based on Spark and YARN in heterogeneous environments [J]. IEEE Transactions on Green Communications and Networking, 2024, 8(2): 635-644.

[16] LI Hongjian, LUO Wei, XIE Wenbin, et al. Adaptive scheduling framework of streaming applications based on resource demand prediction with hybrid algorithms [J]. Journal of Grid Computing, 2024, 22(1): 39.

[17] TARIQ H, DAS O. Execution time prediction model that considers dynamic allocation of Spark executors [C]// Computer Performance Engineering and Stochastic Modelling. Cham, Switzerland: Springer Nature Switzerland, 2023: 340-352.

[18] CHENG Dazhao, WANG Yu, DAI Dong. Dynamic resource provisioning for iterative workloads on Apache Spark [J]. IEEE Transactions on Cloud Computing, 2023, 11(1): 639-652.

[19] SAHIN S, LIU Ling, CAO Wenqi, et al. Memory abstraction and optimization for distributed executors [C]// The 6th International Conference on Collaboration and Internet Computing. Piscataway, USA: IEEE, 2020: 78-87.

[20] LIU Libin, XU Hong. Elasecutor: elastic executor scheduling in data analytics systems [J]. IEEE/ACM Transactions on Networking, 2021, 29(2): 681-694.

[21] KATSOGRIDAKIS P, PAPAGIANNAKI S, PRATIKAKIS P. Execution of recursive queries in Apache Spark [C]// Euro-Par 2017: Parallel Processing. Cham, Switzerland: Springer International Publishing, 2017: 289-302.

[22] XU Yinggen, LIU Liu, DING Zhijun. DAG-aware joint task scheduling and cache management in Spark clusters [C]// International Parallel and Distributed Processing Symposium. Piscataway, USA: IEEE, 2020: 378-387.

[23] CHOI B. Introduction to VMware workstation [M]// CHOI B. Introduction to Python Network Automation Volume I - Laying the Groundwork: The Essential Skills for Growth. Berkeley, USA: Apress, 2024: 231-269.

[24] MOSTAFAEIPOUR A, JAHANGARD RAFSANJANI A, AHMADI M, et al. Investigating the performance of Hadoop and Spark platforms on machine learning algorithms [J]. The Journal of Supercomputing, 2021, 77(2): 1273-1300.

[25] LESKOVEC J, CHAKRABARTI D, KLEINBERG J, et al. Kronecker graphs: an approach to modeling networks [J]. The Journal of Machine Learning Research, 2010, 11: 985-1042.

[26] HAI Mo, ZHANG You, ZHANG Yuejin. A performance evaluation of classification algorithms for big data [J]. Procedia Computer Science, 2017, 122: 1100-1107.

【中文责编：英子；英文责编：木柯】

补充材料