文献引用格式: 霍达,宋利. 基于 Celery 的分布式视频计算处理框架[J]. 电视技术,2016,40(4):12-17.
HUO D,SONG L. Distributed video processing system based on Celery[J]. Video engineering,2016,40(4):12-17.
中图分类号: TP338.8 文献标志码: A DOI: 10.16280/j.videoe.2016.04.003

基于 Celery 的分布式视频计算处理框架

霍 达,宋 利
(上海交通大学 电子工程系 图像通信与网络工程研究所,上海 200240)

摘要: 为了实现能够快速高效的使用计算集群解决视频计算问题,提出一种基于 Celery 的分布式视频处理框架,该框架借鉴了 Hadoop 的架构设计,提出了 API 服务器层,JobTracker 和 TaskTracker 的三层架构,并对其进行了优化使之能够兼容计算资源与存储资源的横向扩展和新的计算应用的纵向扩展。详细描述了框架的架构设计和优化设计。实验数据证明分布式计算框架能够高效地利用多节点实现视频计算。

关键词: 分布式计算; 云计算; 转码

Distributed video processing system based on Celery

HUO Da,SONG Li
(*Image Communication and Network Engineering Institute,Electronic Engineering,Shanghai Jiao Tong University,Shanghai 200240,China*)

Abstract: In view of using computer cluster to solve video processing problem, a distributed video processing system based on celery is proposed in this paper. Using Hadoop's structure as a reference, a three-tier structure is presented, including API server layer, JobTracker layer and TaskTracker layer. This structure enables the system to be compatible with both computing resource and functional extension. In this paper, structure of the system is presented in detail together with optimization. The experimental data indicate that the system is proved to be efficient with multiple calculating nodes.

Key words: distributed computing; cloud computing; transcoding

1 分布式视频计算研究现状

分布式计算由于其易于进行横向扩展的特性,通常被用来处理海量数据。目前也有不少研究试图采用分布式计算框架 Hadoop[1] 进行视频处理,基于 Hadoop 的转码器都采用了类似的架构: 使用 FFmpeg 分割合并视频,同时在 Map 任务中调用 FFmpeg 进行转码。为了提高基于 Hadoop 的视频计算框架的效率,研究者们做了许多工作,比如文献[2]分析了在分布式转码中数据本地化的重要性,据此提出了基于数据本地化的调度策略,文献[3]提出了一种能够负载均衡的文件存储系统,使用该系统能够提高分布式文件系统的工作效率,文献[4]提出了一种任务槽配置算法,能够使得 I/O 资源和计算资源能够并行化利用,有效提高了转码系统的整体效率。这些策略试图集中解决两个问题,其一是工作节点中 I/O 等待和 CPU 计算的并行化设计,其二是提高 HDFS 的读写效率,其优化的思路值得借鉴,但是限于 Hadoop 的架构限制,很难取得进一步的优化。同时基于 Hadoop 的分布式视频处理框架对于功能性扩展限制很大,不能很好地兼容视频计算的一些特点。这两点使得 Hadoop 难以成为通用的视频计算框架。

Celery 是一个使用简单、易于扩展并且可靠的分布式队列,它被设计用以解决软件设计中的任务的异步执行,使用分布式队列可以将软件中各个模块解耦。框架设计中应用分布式队列连接各个模块,保证了系统的伸缩性。

基金项目: 国家自然科学基金项目(61221001)

2 架构设计

2.1 分层设计

分布式计算框架共分三层，分别是 API 服务器层，作业管理层和任务执行层。构建在 Django 上的 API 服务器为终端用户提供 RESTful API 接口，同时调用 JobTracker 的 API，发起、撤销和查询一次作业。JobTracker 管理了每一次作业，并将一次作业切分成许多的任务，并分配给对应的 TaskTracker，TaskTracker 负责执行一次具体的任务。三者在逻辑上行使了不同职责，每两层之间使用 Celery 进行解耦，这保证了 JobTracker 和 TaskTracker 的动态伸缩性。在接下来的小节中，为了表述的方便，本文将由后端发起的一次任务称为一次作业(Job)，将一次具体的视频处理操作称为一个算子(Operator)，一个或多个算子迭代操作为一次任务(Task)。

分布式视频处理系统的分层设计如图 1 所示。

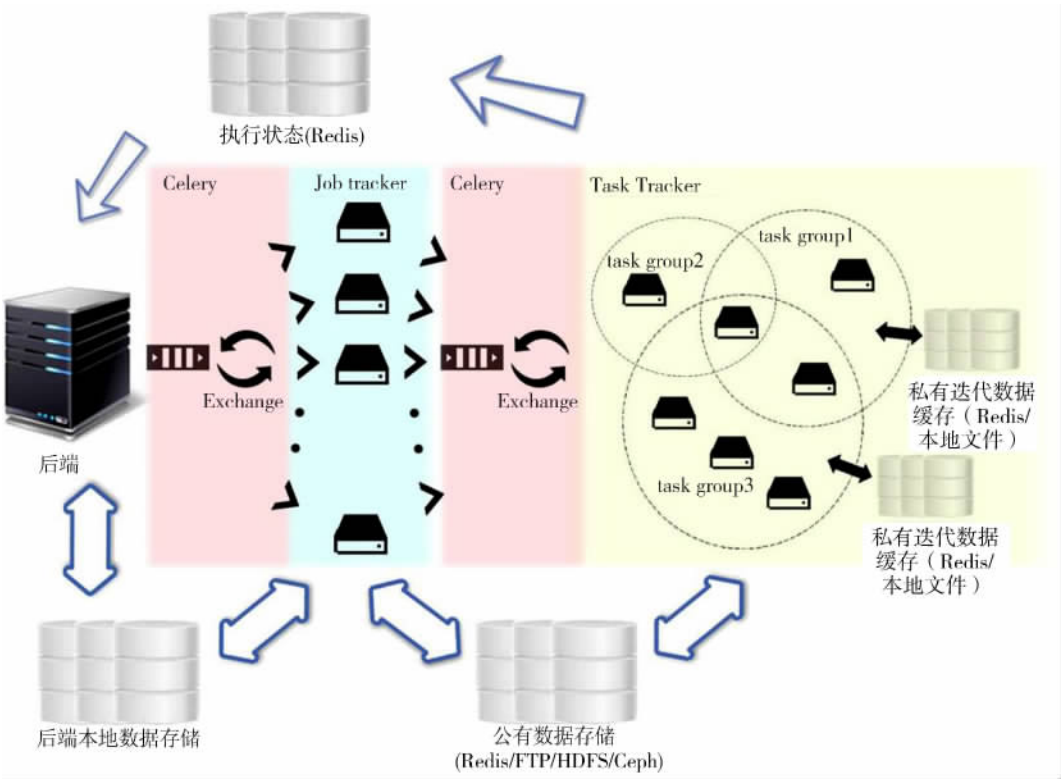

图 1 处理框架层级关系

每层的具体职责如下：

1) API 服务器

API 服务器使用了 RESTful 的架构风格，在 JobTracker 提供的作业功能之上封装了用户管理，对终端用户提供 API，功能涉及到了作业管理和用户管理两类，作业管理包括了作业注册、进度查询、结果查询、历史作业查询和失败原因查询，用户管理使系统支持支持用户私有作业，用户私有文件管理。

2) 作业管理层(JobTracker)

作业管理层具体管理了一次作业，负责的任务有：(1) 分析 API 服务器的调用字符串，将任务拆分成迭代算子序列；(2) 将完整的视频按照固定时间或者固定大小切分成 n 片；(3) 对 n 个分片执行分片上传→处理分片→从缓存服务器下载分片的流水线；(4) 定时查询 n 条流水线的执行状态，整合 n 条分片执行情况并存入 Redis 状态服务器中；(5) 所有流水线完成后合并分片。

3) 任务执行层(TaskTracker)

传统的分布式处理计算框架如 Hadoop，用户自行编写 Map、Reduce 代码，每一个加入到 Hadoop 集群的计算节点都是等价的。在分布式视频处理框架当中，为了执行速度，算子调用第三方库进行运算，框架完成调度，由于软件部署以及硬件支持的原因(比如 GPU)，计算节点之间不等价。为了解决这个问题，计算框架将一些算子集合划分为一个算子族，称为 TaskTracker 集合，算子族之间保持互斥，没有一个算子同时隶属于两个 TaskTracker 集合，一个计算节点由于其部署方式，

可能可以执行一个或多个算子族，同时，一个 TaskTracker 集合之内可能有多个计算节点。算子、计算节点和 TaskTracker 集合三者的关系如图 2、图 3 所示。任务执行的时候由框架选定算子族保证算子可以被执行，为了保证算子序列执行的速度，每个 TaskTracker 集合内的算子将被迭代执行（中间变量被存储在任务处理模块的私有存储中，而非返回到公有数据存储）。TaskTracker 集合中的每个算子都类似于 DirectShow 中的一个 Transform Filter，算子序列迭代结束后，任务处理模块向作业管理层返回状态。一次作业对应一个或多个分片，每个分片对应一个或多个不同的 TaskTracker 集合，一个 TaskTracker 集合对应了一个或多个算子。

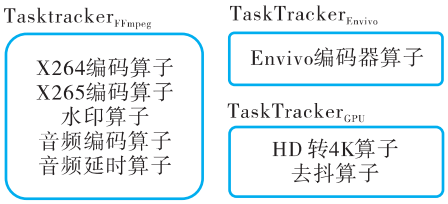

图 2　算子与 TaskTracker 关系示意

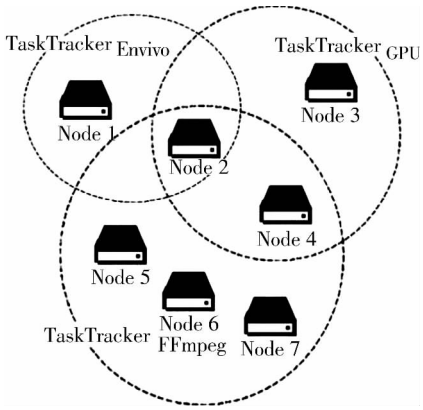

图 3　计算节点与 TaskTracker 关系

2.2　信息流设计

在分布式系统中，本文设计了 3 个流实现分布式系统中的信息流动：

1）控制流

所有的控制流都是由 Celery 进行调度的，控制流的特点是无返回值，并且调用者无法明确得知调用任务执行者的详细状态，细节对于双方不透明。使用 Celery 调度的优点在于，利用其订阅特性，框架可以动态地增加 JobTracker 和 TaskTracker 以保证动态伸缩性。

2）状态流

在分布式任务处理框架中，控制流是单向的，计算框架需要使用状态流描述一个任务的详细状态。状态流与控制流反向，从 TaskTracker 到 JobTracker，最后回到 API 服务器。实现机制如下，TaskTracker 统计算子序列的执行进度，每一个算子的执行时间，如果某个算子执行失败，查询失败原因。JobTracker 查询每一个分片流水线当前所在的 TaskTracker 的状态，汇总，并以 Task 的 UUID 为 Key 存在 Redis 中。API 服务器可以根据 Task 的 UUID 查询 Job Tracker 的状态信息。

3）数据流

在分布式转码中，区别于控制流和状态流，本文将视频文件统称为数据，一次作业中，时间的消耗主要来自于 TaskTracker 执行算子的 CPU 时间和数据流的 IO 时间。对于分布式系统而言，数据流的 I/O 设计决定了框架的执行效率。系统中对于数据的操作有两类：一类是原始文件和完成文件的管理，这些数据以静态文件的形式储存在 API 服务器的硬盘中；另一类是缓存分片文件的缓存，TaskTracker 使用了迭代计算的策略，算子的临时结果会被储存在私有存储当中，不同迭代组通过公有 Redis 交换数据。

图 4 描述了一次典型的作业中，一个分片数据的时序图。假设这个分片需要依次执行 4 个算子，其中前 3 个算子可以被一个 TaskTracker 迭代执行，这 3 个算子的中间结果被放在当前 TaskTracker 所在的物理机上，中间结果储存在硬盘下缓存目录中。在这个实例当中，强制组内算子迭代的设计使 TaskTracker 减少了两次对于公有存储的读写。迭代的设计降低了公有临时存储的并发数，节省了网络 I/O 的时间。当分布式系统逐渐扩展，公有存储节点将承受非常大的负载，其 I/O 将成为系统的瓶颈，为了解决公有存储节点的 I/O 负载问题，Redis 的部署应该使用集群化配置。

2.3　时序图

图 5 描述了一次典型的作业过程。这个例子当中，需要将一个视频先去噪，再编码。视频需要经过两个算子，第一个为去噪算子，第二个为转码算子，执行顺序为去噪→转码，为了简化时序图，图中暂不涉及 TaskTracker 内的算子迭代。分布式系统有一个 JobTracker 实例，3 个 TaskTracker 实例，其中一个 TaskTracker 可以执行转码算子，两个 TaskTracker 可以执行去噪算子。

由 API 服务器发起一次作业，作业请求进入异步消息队列等待空闲的 JobTracker，等到空闲的 JobTrack-

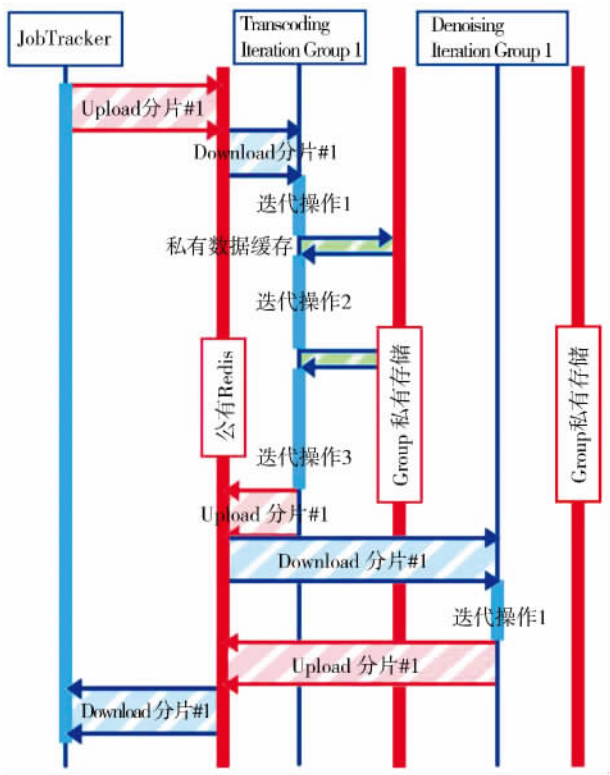

图4　缓存分片时序图

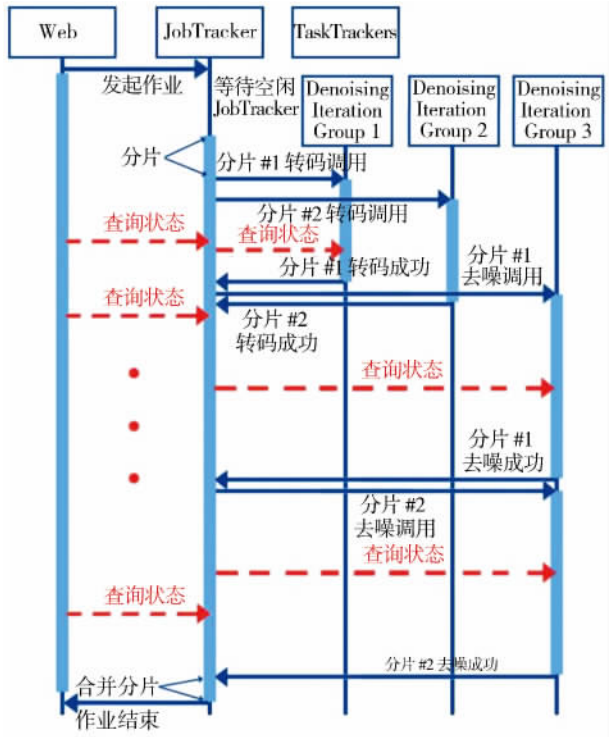

图5　计算节点与 TaskTracker 关系

er 后，JobTracker 开始执行一次作业，首先视频被切分为 2 个分片，两个分片进入上传→去噪 TaskTracker→转码 TaskTracker→下载的流水线。分片#1 和分片#2 依此进入两个空闲的去噪 TaskTracker，并执行算子，#1 和#2 进入去噪模块的时间差是分片上传至公有数据存储的时间，分片#1 执行去噪完毕后，进入转码模块，由于转码 TaskTracker 只有一个，所以两个分片排队执行。两个分片依此转码结束后，回到 JobTracker 执行合并操作。整个过程中，API 服务器在响应上层用户的查询请求时会对 JobTracker 的状态进行查询，JobTracker 对于 TaskTracker 的监控是定时的。

3　性能优化

3.1　JobTracker 的流水线设计

为了解决 JobTracker 的 I/O 问题，JobTracker 使用了流水线的调度方式。图 6 是流水线的时序图。

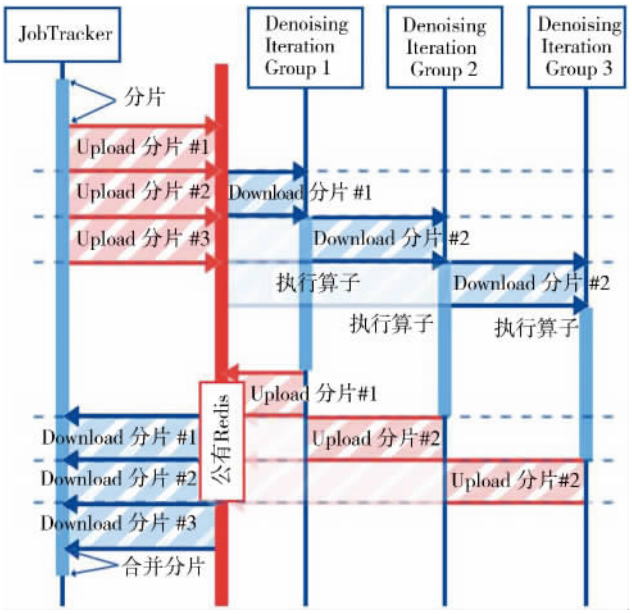

图6　JobTracker 流水线时序图

如果不使用流水线，上传过程将共享带宽，假设分片个数为 n，一次上传时间为 T_{upload}，则一次 Job 中耗的时间为

$$(n-1)\,T_{upload} \tag{1}$$

如果分片处理时间相仿，则下载分片时会出现拥堵现象。消耗时间有可能会更高。使用流水线解决了 JobTracker 内的同步问题，JobTracker 间的并行机制比较复杂，且单台物理机的 I/O 上限难以突破，为了解决系统扩展时的 I/O 瓶颈问题，可以使用多 API 服务器实例 + Nginx 反向代理的方式进行负载均衡。

3.2　算子的切分粒度设计

TaskTracker 执行 JobTracker 分配的算子序列，算子

的设计有以下两种原则，具体在实现中使用哪一种原则取决于应用的倾向:

1) 为了提高执行效率，分布式框架中的算子应该维持一个比较大的任务切分粒度。此时的算子为比较复杂的计算，或者是多个简单的算子组成一个比较大的算子，迭代在算子内部迭代执行。

2) 为了提高扩展性，此时算子切分粒度最小，每个算子只执行一件任务，任务在 TaskTracker 内迭代执行，虽然效率比算子内部迭代执行低，但是算子之间的组合清晰，对于任务的扩展友好。

TaskTracker 运行在计算节点上，最大化利用计算节点资源是 TaskTracker 设计的主要问题，TaskTracker 的执行耗时主要有，算子执行，消耗 CPU 计算资源; 中间结果的 I/O，消耗硬盘 I/O 资源; 分片下载与上传，消耗网络 I/O 资源; 在实际部署时，单物理节点部署应启动多 TaskTracker 实例，实例的个数以达到 CPU，硬盘 I/O，网络 I/O 中的任意一个瓶颈为准，这区别于物理机性能，算子类型。

3.3 失效转移与负载均衡

Celery 是一个分布式系统，不存在主控节点，TaskTracker 和 JobTracker 实例最多，最有可能发生单点故障，单点故障的可以通过失效转移实现，机制如下: 服务器定时对正在处理的 JobTracker 进行心跳检测，失效则转移，JobTracker 层也需要对所有正在处理的 TaskTracker 进行心跳检测，失效则转移，此外 JobTracker 还负责失效缓存分片的垃圾回收。Celery 底层依赖 RabbitMQ，RabbitMQ 的失效转移通过 Mirror Queue 机制实现，该机制保证了队列中的消息在每个主备节点当中都有一份拷贝，所以当一个节点失效的时候，整个集群仍然有效，代价在于拷贝消息会带来比较大的性能损失，系统的吞吐量会有所下降。考虑到 API 服务器可能出现单点故障，以及 Web API 服务器的 I/O 瓶颈问题，在高负载、高并发的生产环境中可以使用多 Web API 服务器实例 + 多 Redis 集群部署方案，并使用 Nginx 反向代理，这样的部署方式可以同时避免 API 服务器以及 Celery 节点单点故障带来的影响，性能上也更为出色。

4 实验结果

本节通过实验研究分布式视频处理框架的性能，实验环境为 HP Z820 工作站 3 台，CPU Intel Xeon E5-2698 v2 2.7 GHz，内存 32 Gbyte。本文挑选了一个原始分辨率为 3 840 × 2 160，码率 140 kbit/s，长度为 2 min 13 s，帧率为 30 f/s(帧/秒) 的视频作为基准视频，以计算复杂度为区分，确定了 3 个计算复杂度各异的任务:

1) Benchmark A: 分辨率 640 × 360，码率为 8 000 kbit/s，使用 libx264 单线程进行压缩，压缩参数: 分辨率 640 × 360，码率为 8 000 kbit/s，直接调用 libx264 进行压缩时耗时为 99.89 s。

2) Benchmark B: 分辨率为 1 280 × 720，码率为 20 000 kbit/s，使用 libx264 单线程进行压缩，压缩参数: 分辨率 1 280 × 720，码率为 20 000 kbit/s，直接调用 libx264 进行压缩时耗时为 380.34 s。

3) Benchmark C: 分辨率为 1 920 × 1 080，码率为 40 000 kbit/s，使用 libx264 单线程进行压缩，压缩参数: 分辨率 1 920 × 1 080，码率为 40 000 kbit/s，直接调用 libx264 进行压缩时耗时为 851.72 s。

本文选定 Benchmark C 作为转码任务，测试了随着工作节点的变化，执行的时长，结果如图 7 所示，可以看到，随着工作节点的增加，计算耗时逐渐降低。

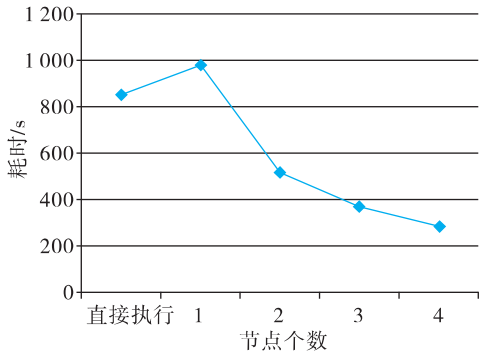

图 7 工作节点数改变时计算时间的变化

相比于与直接执行算子，调用计算框架进行计算时会有额外的时间损耗，主要来自于分片的上传下载、分片切割、合并，单个计算节点的工作效率理论上只能接近 100%。下面本文使用计算的工作效率进行分析: 规定当直接调用算子时，效率为 1，分布式计算时，以直接执行算子处理相同任务的时间为基准，单节点效率 μ 的计算公式如下

$$\mu = \frac{T_{\text{overall}}}{n_{\text{node}} \cdot T_{\text{direct}}} \tag{2}$$

式中：n_{node}为节点数；$T_{overall}$为分布执行时间；T_{direct}为直接执行算子耗时。

表1描述了不同Benchmark下工作节点数关系与效率值的关系，可以看到两个趋势：

1）计算越复杂，效率越高，这主要是因为影响效率值的主要因素是I/O损耗，在I/O损耗一定时，如果计算越复杂，CPU计算时间长，效率提升，这也是分布式计算框架的主要应用方向，即高CPU负载的计算任务。

2）随着计算节点个数的增加，效率逐渐降低，这主要是因为实验环境的网络负载达到瓶颈，I/O时间延长。

表1　不同Benchmark下工作节点数与效率值

Benchmark	不同节点数下的执行效率/%			
	直接运行	1	2	3
Benchmark A	100	82.38	78.54	69.11
Benchmark B	100	87.57	83.65	76.96
Benchmark C	100	86.96	82.90	76.67

表2描述了研究分片时间与分布式系统工作效率的关系，从结果显示，分片时间对于工作效率的影响是比较小的。分片时间决定了子任务的任务切分粒度，当分片时间比较短的时候，任务粒度较小，处理时间比较稳定，但是分片过多引起I/O效率有下降的趋势。当分片时间比较大的时候，任务粒度过大，有可能导致有些工作节点的闲置，导致计算框架整体效率下降。总体来说，分片时间与分布式计算框架的效率关系并不是太大，选择2~20 s的分片大小均可以取得一个相对稳定的工作效率。

表2　分片时间改变时效率执行时间的变化

节点	不用分片时间下的执行时间/s				
	2 s	4 s	8 s	16 s	32 s
1节点	121.26	120.23	120.49	114.70	119.39
2节点	63.59	62.89	64.06	60.60	69.45
3节点	48.18	48.41	47.51	47.97	60.13
4节点	38.69	38.25	37.90	38.89	44.45

表3描述了TaskTracker使用本地存储和使用远端存储对于单节点效率的影响。可以看到，无论使用本地节点还是远端节点，都有着节点数增大，效率降低的问题，同时由于I/O时间的增加，使用远程缓存的工作效率会下降7个百分点左右。所以使用迭代的策略是很有必要的。

表3　使用公有存储和私有存储对效率值的影响

存储方式	不同节点数下的执行效率/%			
	1	2	3	4
私有存储	86.96	82.90	76.67	74.98
公有存储	80.91	76.55	69.57	67.04

5　小结

本文提出了一种基于Celery的分布式视频处理框架，该框架借鉴了Hadoop的架构设计，并对进行了优化使之能够兼容计算资源与储存资源的横向扩展和新的计算应用的纵向扩展。此外，本文还提出了对于计算框架的一些优化机制，实验数据证明分布式计算框架能够高效地利用多节点实现视频计算。

参考文献：

[1] DIAZ-SANCHEZ D，MARIN-LOPEZ A，ALMENAREZ F，et al. A distributed transcoding system for mobile video delivery [C] // 2012 5th Joint IFIP Wireless and Mobile Networking Conference (WMNC). [S. l.]: IEEE，2012: 10-16.

[2] YOO D，SIM K M. A comparative review of job scheduling for MapReduce [C] //2011 IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS). [S. l.]: IEEE，2011: 353-358.

[3] YE X，HUANG M，ZHU D，et al. A novel blocks placement strategy for Hadoop [C] //2012 IEEE/ACIS 11th International Conference on Computer and Information Science. [S. l.]: IEEE，2012: 3-7.

[4] 陈珍. 基于MapReduce的海量视频转码系统优化机制[D]. 武汉：华中科技大学，2013.

作者简介：

霍　达，硕士生，主要研究方向为视频传输协议和分布式视频处理框架；

宋　利，副教授，研究方向为图像处理、视频编码。

责任编辑：时　雯　　收稿日期：2015-12-15