

Redis 分布式缓存技术在 Hadoop 平台上的应用

姚经纬¹, 杨福军²

(1. 江南大学 物联网工程学院, 江苏 无锡 214122;

2. 中国空气动力研究与发展中心 计算空气动力研究所, 四川 绵阳 621000)

摘要:在使用 Hadoop 进行大规模数据分析时,经常会遇到的一个较为典型的问题就是共享数据的快速访问问题。该类问题存在的场景很多,如网页排名算法、最小错误率训练算法、最大期望算法等。虽然已有关于此类问题的解决方案,但实际取得的效果却不尽如人意。为此,提出了使用 Redis 内存数据库作为分布式缓存,以解决 Hadoop 中共享数据访问的问题。验证实验结果表明,Redis 分布式缓存的吞吐率与集群规模有较好的线性关系,所提出的方法能够较好地解决 Hadoop 任务对共享数据的访问问题,同时也为其他大规模共享数据访问的问题提供了简便的解决思路。Redis 作为开源的商业化工具,使得所提出的方法具有较好的适用性,可为科研以及生产实践中遇到的同类问题提供一种较为通用的解决方案。

关键词: Redis; 分布式缓存; Hadoop; MapReduce

中图分类号: TP311.5

文献标识码: A

文章编号: 1673-629X(2017)06-0146-05

doi: 10.3969/j.issn.1673-629X.2017.06.030

Application of Redis Distributed Caching Technology in Hadoop Framework

YAO Jing-wei¹, YANG Fu-jun²

(1. School of IoT Engineering, Jiangnan University, Wuxi 214122, China;

2. Computational Aerodynamics Institute, China Aerodynamics Research and Development Center, Mianyang 621000, China)

Abstract: In the scene of large scale data analysis with Hadoop, rapid accessing to shared resources is a typical problem that has not been satisfactorily solved so far. Examples of such problem include page rank algorithm, minimum error-rate training algorithm, expectation maximization algorithm and so on. Although solutions to such problems have existed, the actual effect is not satisfactory. Thus, an open-source distributed in-memory database, Redis, has been explored to provide high-throughput access to shared resources in Hadoop. Experimental results illustrate that Redis has the characteristic of linear increase in throughput with respect to cluster size so that it can provide a general-purpose solution for rapid accessing to shared resources in Hadoop cluster, and that it has provided an easier implementation of algorithms that has not been satisfactorily solved at large scale with Hadoop. Meanwhile, the use of Redis, the commercial-grade open-source tool, implies that the proposed solution has been easily adapted in both research and production environments.

Key words: Redis; distributed caching; Hadoop; MapReduce

0 引言

随着信息技术的飞速发展,互联网数据量也呈现出爆炸式增长,进入了大数据时代^[1]。对于日益增长的海量数据处理,传统的数据处理方式已无法支持如此庞大的数据量,因而云计算技术应运而生。在诸多云计算平台中,Hadoop 凭借其开源、廉价等优势,在大数据的存储和处理等方面应用广泛^[2],很多互联网企

业,如亚马逊、阿里巴巴、中国移动等都纷纷使用 Hadoop 作为自己的数据处理平台。Hadoop 的核心主要是 HDFS(分布式文件系统)和 MapReduce 分布式数据处理框架;除此之外,还有很多基于 Hadoop 的工具,如: HBase 分布式数据库、Hive 数据仓库分析工具、Spark 流式数据处理框架等^[3]。

Hadoop 的出现大大简化了分布式程序设计,使用

收稿日期: 2016-07-08

修回日期: 2016-10-11

网络出版时间: 2017-04-28

基金项目: 工信部高技术船舶项目(2016[26])

作者简介: 姚经纬(1991-),男,硕士,研究方向为计算机应用技术、软件工程。

网络出版地址: <http://www.cnki.net/kcms/detail/61.1450.TP.20170428.1703.060.html>

者只需要简单地将数据处理应用分解为 Mapper 和 Reducer,就可以使之运行在 Hadoop 集群上,而不用关心各节点之间如何通信、如何传递数据等底层实现^[4]。然而,Hadoop 在实际使用中仍然存在一系列问题亟待解决,任务节点如何快速访问海量共享数据则是其中一个较为典型的问题。

针对 Hadoop 任务节点如何快速访问海量共享数据的问题,基于对典型实例的阐述和分析,提出了使用 Redis 作为分布式缓存解决共享数据的访问问题。并以 PageRank 算法为例,研究分析了如何使用 Redis 解决 PageRank 算法中网页得分数据的存储和访问问题,通过实验对该方法的可行性和效率进行了验证,并为其他相似问题的解决提供了思路。

1 MapReduce 和 Redis 简介

1.1 MapReduce 框架

MapReduce^[5]最先是由 Google 公司的 Jeffrey Dean 和 Sanjay Ghemawat 提出的一种用于解决大数据并行计算问题的编程模型。后来 Apache 基金会的 Doug Cutting 在 Hadoop 项目中实现了该模型并将其开源,大大推动了 MapReduce 框架的研究和发展。MapReduce 程序以键值对<key,value>的形式进行输入输出,其具体执行步骤如下:

(1)任务分配:JobTracker 将整个作业分解为多个任务,并分配到任务数据所在的节点上,由各节点的 TaskTracker 负责任务执行。

(2)Map 阶段:任务节点将当前节点上的数据块解析为键值对< k_1, v_1 >输入到 Map 函数中,Map 函数对输入进行过滤和转换,再以集合 $\text{list}(<k_2, v_2>)$ 的形式输出并保存为中间结果。

(3)洗牌:将键 k_2 相同的记录发送到同一个 Reduce 任务节点上。

(4)Reduce 阶段:任务节点将所有键 k_2 相同的记录以键值对< $k_2, \text{list}(<v_2>)$ >的形式输入到 Reduce 函数中,Reduce 函数对输入进行计算处理后,再以结果集 $\text{list}(<k_3, v_3>)$ 的形式输出。

(5)作业结束:当所有 Map 任务和 Reduce 任务结束后,整个作业结束。

在 MapReduce 编程中,编程人员只需根据业务编写 Map 函数和 Reduce 函数就可以实现比较复杂的并行计算作业,而不用关心各节点之间如何通信、如何传递数据等底层实现,大大简化了并行编程的复杂度。MapReduce 程序还具有很好的可扩展性,在大多数情形下,它不仅随着数据规模的扩大表现出持续的有效性,而且在性能上能随着节点数的增加保持接近线性的增长。然而,MapReduce 还具有低成本和高可

靠性等众多特征,使其从一开始就受到了学术界和商业界的极大关注^[4]。

1.2 Redis 内存数据库

Redis^[6]是一种基于内存的 Key-Value 数据库产品,是由远程字典服务(Remote dictionary server)取名而来。它支持多种数据类型的存储:字符串(string)、链表(list)、集合(set)、有序集合(zset)和哈希类型(hash),并且各种类型都支持丰富的操作,其中大多都支持原子操作。为了保证数据存取的效率,数据都保存在内存中;Redis 还提供了对持久化的支持,它可以定期将更新的数据异步写入磁盘,同时不影响继续提供服务;在此基础上,还实现了主从复制,这对预防单点故障和提高负载能力有很大帮助。Redis 的出现,在很大程度上弥补了 Memcached 的不足,它不仅支持更加丰富的数据类型和操作,而且在读写效率上也比 Memcached 更胜一筹。根据 Redis 官方测试数据,Redis 写入速率为 198 412.69 条/s,读速率为 198 019.80 条/s^[7],相比 Memcached 要高出数倍。Redis 具有如此多的优秀特性,使其从一开始就受到了广泛关注,Redis 可以适用于多种不同的应用场景,很多大型互联网企业的后台服务中都有使用,并且存在不少成功应用的范例。

虽然 Redis 读写性能非常优秀,但是因为内存容量的限制,仅使用单台服务器一般是不够的,这就需要使用集群的形式进行水平扩容。在旧版 Redis 中通常使用客户端分片来构建集群,但这种方式有很多缺点,比如维护成本高,增加、移除节点较繁琐等;但 Redis 3.0 版的发布解决了这一问题,因为它增加了对集群的原生支持。Redis 集群采用无中心架构,各节点间使用 Gossip 协议进行通信;在对数据的分配上使用预分桶的策略,将每个键的键名有效部分使用 CRC16 算法计算出散列值,然后对 16 384 取余,使得每个键都可以被分配到预先分配好的 16 384 个插槽,进而在对应的节点中进行处理;集群具有较高的可用性,它采用主-从形式,确保当主节点失效后可以将一个从节点转变为主节点,以此确保集群的完整性和可用性^[8]。Redis 集群的这些特性使得能够很方便地将其作为分布式缓存使用。

2 问题探讨

Hadoop 在生产实践中被广泛应用于大数据的存储和处理,并且存在很多成功应用的典范。但是在实际应用中暴露出一些问题,其中一个较为典型的任务节点如何快速访问海量共享数据的问题。存在该类问题的算法和情景不在少数,这里仅列举三个典型对该类问题进行简单阐述。

2.1 网页排名算法

网页排名算法 (PageRank)^[9-10] 是由 Google 创始人 Sergey Brin 和 Lawrence Page 提出的用于在搜索引擎上对网页进行排名,以此体现网页重要性的一种算法。该算法初始时为每个网页设置一个得分,经过多次迭代不断更新各网页的得分,最终各网页得分收敛时算法结束。在每次迭代中,都需要根据每个网页的得分给所有链出网页打分,每个网页根据所有链入网页给出的打分,计算并更新自己的得分。在 Hadoop 上运行该算法时,网页得分数据是所有任务的共享数据,在每个任务中都需要获取和更新网页得分数据,因此网页得分数据的访问效率会直接影响算法的执行效率。而且在实际应用中,网页得分数据往往会达到数百亿条,因此,如何存储和访问网页得分数据则是接下来所主要讨论的问题。

2.2 最小错误率训练算法

最小错误率训练算法 (MERT)^[11] 是由 Franz Josef Och 提出的一种在机器翻译中以翻译质量为优化目标,以此调节对数线性模型参数的算法。该算法初始时生成翻译候选和对应特征权重,经过多次迭代不断对其进行更新,直到不再产生新的翻译候选时算法结束。每次迭代中使用解码器对翻译候选进行解码,生成新的翻译候选与之前的合并,并在扩展的候选集上重新调整特征权重。在 Hadoop 上运行 MERT 算法时,特征权重数据是所有任务的共享数据,其访问效率会直接影响到算法的执行效率。实际应用中,特征权重数据也可能会达到数十亿条,那么又该如何解决数据的存储和访问问题。

2.3 最大期望算法

最大期望算法 (EM)^[12] 是由 Arthur Dempster 等提出的在已知部分相关变量的情况下,寻找未知变量的最大似然估计或最大后验估计的算法,在数据挖掘领域的聚类算法中应用广泛。以基于高斯混合模型的 EM 算法为例,它分为两个阶段交替执行直到模型参数趋于收敛:

(1) 步骤:根据数据集和模型参数计算对数似然函数的条件期望;

(2) 更新模型参数,使对数似然函数期望最大化。

在 Hadoop 上运行 EM 算法时,模型参数为所有任务所共享,其访问效率同样会直接影响算法的执行效率。同样,当模型参数数据量过大时,又该如何解决数据的存储和访问问题。

上述三类问题都涉及到任务节点如何访问共享数据这一共性问题。虽然 Hadoop 提供了分布式文件缓存机制,可以将共享文件拷贝到每个任务节点并装载到内存中以实现数据的共享,该方法确实可以在一定

程度上解决上述问题;但是当共享文件过大无法正常装载到任务节点的内存中时,该方法就不再适用,这在实际应用中并不罕见;况且,对每一个任务节点来说,它所需要的数据可能仅仅是全部共享数据的一小部分,这种情况下将全部共享数据拷贝到任务节点上不仅浪费网络和内存资源,而且还可能拖慢任务的执行。因此,提出了使用 Redis 内存数据库作为分布式缓存,实现在 Hadoop 任务节点间快速获取共享数据的方法,从而更好地解决上述问题。

3 解决方案

根据讨论中提及的三个问题,需要一种能够在 Hadoop 任务节点间快速获取共享数据的机制,并且必须满足以下条件:

- (1) 键必须保证全局唯一性;
- (2) 必须能够支持大规模的数据存储;
- (3) 必须确保在高并发量前提下数据访问的高效性。

Redis 内存数据库的特性刚好满足上述需求^[13],因此,提出子在 Hadoop 中引入 Redis 分布式缓存来解决共享数据的存储和访问问题。

使用 Redis 作为分布式缓存,需要确保 Hadoop 集群中各节点都能同等地访问 Redis 中存储的数据,因此,采用图 1 的架构方式。这种将 Hadoop 集群与 Redis 分布式缓存直接相连的方式,不仅在实现上比较简单,而且也最大程度地保证了数据存取的效率。对于分布式缓存的使用,一般分两步进行:

首先将 HDFS 上的共享数据加载到分布式缓存中。这一步并不需要用到 Reduce,因此发起一个只有 Map 阶段的作业即可完成,各 Map 任务可以并行地读取数据,并保存到分布式缓存中。

当分布式缓存数据准备完成后,启动需要执行的 MapReduce 作业。各任务节点在初始化时使用 Jedis 客户端建立起 Redis 集群的连接,这样,在任务执行中需要访问缓存时就可以随时通过连接读写共享数据。

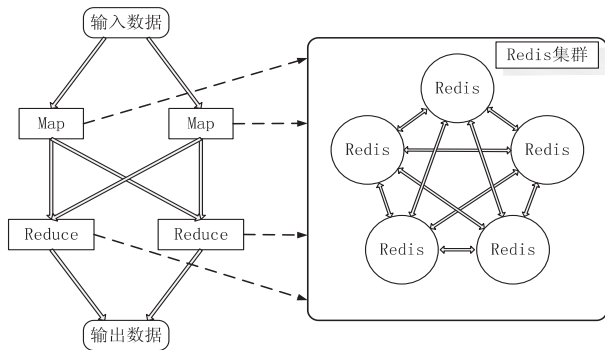


图 1 Hadoop 与 Redis 分布式缓存架构图

为了进一步阐述问题,并验证 Redis 作为分布式缓存的性能,以网页排名算法为例,阐述 Redis 分布式缓存在 Hadoop 任务中的应用。在实例中使用原始的网页排名算法进行阐述,一方面,研究的主要目的在于使用 Redis 分布式缓存解决大规模共享数据问题,而非仅仅论述网页排名算法的优化问题,对网页排名算法的优化不作为研究重点;另一方面,对网页排名算法的优化大都集中于如何减少迭代次数或如何在每次迭代中减少需要处理的数据等方面,优化后的算法仍可能出现上述问题,而原始的网页排名算法具有很好的代表性,能够较简明地说明问题。

网页排名算法计算网页排名基于以下两个基本假设:

(1)数量假设:一个具有较多链入的网页会有较高分。

(2)质量假设:一个得分较高的网页能够给其链出的网页打出较高的分数。

根据这两个假设,可得^[9]:

$$\text{PR}(p_i) = \frac{(1 - d)}{N} + d \sum_{p_j \in M(p_i)} \frac{\text{PR}(p_j)}{L(p_j)}$$

(1)

其中, p_i 和 p_j 表示两个不同的网页; $\text{PR}(p_i)$ 和 $\text{PR}(p_j)$ 分别表示 p_i 和 p_j 的得分; $M(p_i)$ 表示所有链入 p_i 的网页集合; $L(p_j)$ 表示 p_j 链出的网页数目; N 表示全部网页数目; d 表示阻尼系数(表示用户到达某网页后继续向后浏览的概率,一般取 0.85)。

网页排名算法计算网页得分是一个迭代计算的过程。初始时赋予每个网页相同的得分,在之后的每次迭代中,使用式(1)更新得分,直到所有网页得分稳定时算法终止。

使用 Redis 作为分布式缓存,在 Hadoop 上实现网页排名迭代算法的步骤如下:

(1)原始数据的预处理。对原始数据进行处理,生成符合格式要求的网页链接数据文件,并保存到 HDFS 中。网页链接数据文件的每行第一列表示当前网页链接地址,后面的各列表示当前网页所有链出的网页地址,各列以 Tab 键分隔,后文出现的网页链接数据,如无特别说明,都使用该格式。

(2)初始化网页得分数据并保存到 Redis 中。启动一个只有 Map 阶段的作业用来读取网页链接数据,Map 函数中,将当前的键(url)和初始化得分(score)以键值对<url,score>的形式保存到 Redis 分布式缓存中。Map 函数的伪代码如下:

```
1 //key:当前网页的链接地址;value:以 Tab 键分隔的所有链出地址
2 Map(key,value) {
3     init = 0.5; //默认初始得分
```

```
4     setToRedis(key,init); //将键值对保存到 Redis 分布式缓存中
5 }
```

(3)使用一次 MapReduce 作业完成网页排名算法的一次迭代。在每次迭代的 Map 函数中,从分布式缓存中获取当前网页得分(score),将该得分平均分配给各链出地址(url,共 n 个)作为对该链接的打分,并以键值对<url,score/ n >输出。Map 函数的伪代码如下:

```
1 //key:当前网页的链接地址;value:以 Tab 键分隔的所有链出地址
2 Map(key,value) {
3     //根据键从 Redis 分布式缓存中获取相应的值
4     score = getFromRedis(key);
5     urls = value.split("\t"); //将 value 以 Tab 键分割得到数组
6     for(url ;urls) {
7         context.write(url,score/urls.length); //Map 的输出
8     }
9 }
```

在每次迭代的 Reduce 函数中,将其他网页给本网页(url)的打分计算汇总后得出本网页的得分(score),并以键值对<url,score>的形式保存到 Redis 分布式缓存中。Reduce 函数的伪代码如下:

```
1 //key:本网页的链接地址;values:其他网页给本网页的打分集合
2 Reduce(key,values) {
3     d = 0.85; //阻尼系数
4     //sum(values):对 values 集合进行求和
5     score = (1 - d) + d * sum(values);
6     setToRedis(key,score); //将键值对保存到 Redis 分布式缓存中
7 }
```

4 实验结果及分析

实验中使用 9 台普通 PC,每台 PC 配置如下:3 GB 内存,酷睿 2 双核 CPU,500 GB 硬盘,Ubuntu 14.04 操作系统。实验使用 Apache Hadoop 1.2.1,其中 1 台作为 NameNode 和 JobTracker,其他 8 台作为 DataNode;Redis 版本 3.0.7,分别搭建在 8 台 DataNode 上,构成一个 8 节点的 Redis 集群作为分布式缓存。

实验数据使用网络爬虫从互联网上爬取,经过过滤和预处理后得到符合格式要求的网页链接数据。链接数据共包含 36 323 864 个网页节点,大小约 37 GB。实验按照第三节中的步骤进行,作业执行时间使用 4 次迭代的平均时间计算。在每次迭代中,需要读取缓存 36 323 864 次,写入缓存 36 323 864 次,共计访问缓存 72 647 728 次。

实验结果如表 1 和图 2 所示。

表 1 Redis 分布式缓存实验结果

| DataNode 数目 | 作业执行时间/s | 处理记录数目 | 缓存访问次数 | 缓存吞吐率/(q/s) |
|-------------|----------|------------|------------|-------------|
| 2 | 3 712 | 36 323 864 | 72 647 728 | 19 571 |
| 4 | 2 648 | 36 323 864 | 72 647 728 | 27 435 |
| 8 | 1 823 | 36 323 864 | 72 647 728 | 39 850 |

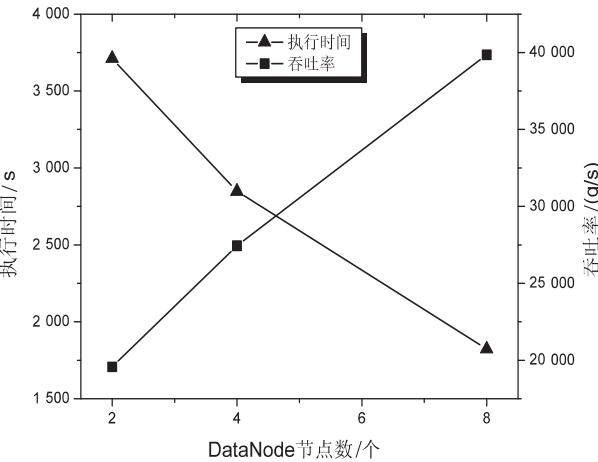


图 2 节点数与执行时间和吞吐率关系图

从图 2 中可以看出,随着 Hadoop 集群节点数的增加,作业执行所需的时间在减少。这是因为随着任务并发量的增大,相同时间内能够处理更多的数据,因此作业执行所需的时间也会相应减少。与此同时,随着 Hadoop 集群节点数的增加,Redis 分布式缓存的吞吐率接近直线增加($R=0.996$),也就是说,Redis 分布式缓存的吞吐率与并发访问量有较好的线性关系。

为了对使用了 Redis 分布式缓存的作业与普通 MapReduce 作业的执行效率进行比较,实现了 PageRank 算法的 MapReduce 程序^[10,14]:首先启动一个 Hadoop 作业,将网页链接文件和链接得分文件同时输入,通过 Reduce 汇总后计算得到该网页对其他网页的打分,作为中间文件输出到 HDFS 上;然后启动另一个 Hadoop 作业,将中间文件作为输入,通过 Reduce 汇总后计算得到每个网页的打分并输出。这样就完成了 PageRank 算法的一次迭代。使用与上述实验同样的数据集和集群进行实验,结果如表 2 和图 3 所示。

表 2 使用 Redis 分布式缓存与普通 MapReduce 作业的实验结果

| DataNode 数目 | 使用 Redis 分布式缓存 作业执行时间/s | 普通 MapReduce 作业 执行时间/s |
|-------------|----------------------------|---------------------------|
| 2 | 3 712 | 4 631 |
| 4 | 2 648 | 3 472 |
| 8 | 1 823 | 2 089 |

从图 3 中可以看出,使用 Redis 分布式缓存后,PageRank 作业的执行效率与普通基于 MapReduce 的作业执行效率相比有所提高,这主要是因为从内存中

读取数据与从硬盘读取数据相比更加高效的缘故;其次,直接从 Redis 中获取共享数据与采用其他替代的方式间接获取共享数据相比,不仅降低了程序的复杂度,而且更加简便高效。

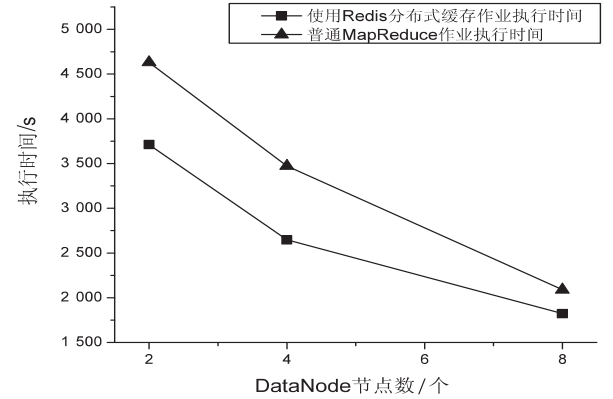


图 3 使用 Redis 分布式缓存与普通 MapReduce 的作业执行时间

以上结果表明,Redis 集群在高并发的情况下仍然能够保持优良的性能,因此 Redis 能够很好地与 Hadoop 平台相结合,作为在任务执行过程中高效获取共享数据的分布式缓存,解决共享数据的存储问题。而且,Redis 集群本身还具有很好的可扩展性,可以通过增加节点数目扩大集群的容量,而且在性能上也能保持接近线性的增长,这一特性使得日后数据规模扩大后可以比较简单地通过增加节点的方式实现扩展,而不用对源程序作任何修改。同时,Redis 作为成熟的商业产品,具有使用简单、易于推广的特点,使得该方案能够比较容易地运用于实践中,为 Hadoop 任务中共享数据的访问提供一种简单、高效的解决方案。

关于所提到的最小错误率训练算法和最大期望算法的问题,也可以使用与上面所提到的网页排名算法类似的解决方案,即将共享数据加载到 Redis 分布式缓存中,这样在任务执行时各任务节点就可以随时访问分布式缓存中的数据,此处就不再一一赘述。

综上所述,Redis 分布式缓存具有性能高、扩展性好、使用简单等特点,因此可以作为在 Hadoop 任务中访问共享数据的有力工具,为相关问题提供一种简单高效的解决方案。虽然 Redis 作为分布式缓存在性能上已经足够高效,但是仍有可以进一步优化之处,比如:使用批量提交请求的方式减少交互次数,使用异步的请求方式提高并行度,使用 UDP 协议加快访问速度,实现 Redis 集群负载均衡以提高效率……这些 Redis 性能优化问题值得进一步深入研究。

5 结束语

为了解决实际应用中 Hadoop 任务节点快速访问
(下转第 155 页)

法。该方法综合考虑系统软硬件,建立了能耗模型,并通过严格的数学推导公式预测整个系统的能耗。以某传感器节点为实例的验证结果表明,所提出的方法有效、可行。该方法在嵌入式系统设计初期对系统架构的能耗分析具有很好的参考价值,方便设计人员根据能耗要求及时调整系统架构,节省设计的时间、物力成本。

参考文献:

[1] 郭 兵,沈 艳,邵子立. 绿色计算的重定义与若干探讨 [J]. 计算机学报,2009,32(12):2311-2319.

[2] 赵 霞,郭 耀,陈向群. 软件能耗优化技术研究进展[J]. 计算机研究与发展,2011,48(12):2308-2316.

[3] Tiwari V, Maliks S, Solfe A. Power analysis of embedded software: a first step towards software power minimization [J]. IEEE Transactions on Very Large Scale Integration, 1994, 2(4):437-444.

[4] 杨志斌,皮 磊,胡 凯,等. 复杂嵌入式实时系统体系结构设计与分析语言: AADL[J]. 软件学报,2010,21(5):899-915.

[5] Blouin D, Senn E. CAT: an extensible system-level power consumption analysis toolbox for model-driven [C]//Newcas conference. [s. l.]:[s. n.], 2010:33-36.

[6] Senn E, Douhib S, Blouin D, et al. Power and energy estima-

tions in model-based design [M]. Netherlands: Springer, 2009:3-26.

[7] 罗 增. 一种基于 AADL 语言的移动软件能耗评估方法 [D]. 福州:福建师范大学,2015.

[8] Wu Zhehui. Petrinetwork introduction [M]. Beijing: Mechanical Industry Press, 2006:47-64.

[9] Zhang Hongmei, Liu Fei, Yang Ming, et al. Simulation of time petri nets [C]//Fourth international conference on system science, engineering design and manufacturing. Guiyang: [s. n.], 2013:26-27.

[10] Hugues J, Zalila B, Pautet L, et al. From the prototype to the final embedded system using the Ocarina AADL tool suite [J]. ACM Transactions on Embedded Computing Systems, 2008, 7(4):1-25.

[11] Feiler P H, Gluch D P. Model-based engineering with AADL [M]. [s. l.]:[s. n.], 2012.

[12] Girault C, Valk R. Petri nets for systems engineering [M]. [s. l.]:Springer-Verlag, 2003.

[13] 吴育春,李蜀瑜. 基于时间 Petri 网的 AADL 模型[J]. 计算机技术与发展, 2014, 24(2):88-91.

[14] 谢和平. 基于排队 Petri 网的感知节点能耗建模技术研究[D]. 哈尔滨: 哈尔滨工业大学, 2013.

[15] 周海鹰,徐 杰,高 妍,等. 基于状态转移的感知节点能耗模型研究与设计 [J]. 计算机应用研究, 2012, 29(9):3432-3436.

(上接第 150 页)

较大规模的共享数据的相关问题,以在 Hadoop 集群中引入 Redis 分布式缓存的方式,为该类问题提供了一种简单、高效的解决方案。实验结果表明,Redis 分布式缓存存在高并发访问时仍具有优异的性能,同时还具有扩展性好、使用简单等特点,这些使得该方案能够很好地与实践相结合,解决 Hadoop 任务中共享数据的访问问题。

参考文献:

[1] Viktor M S, Cukier K. 大数据时代 [M]. 杭州:浙江人民出版社, 2013.

[2] 严霄凤,张德馨. 大数据研究 [J]. 计算机技术与发展, 2013, 23(4):168-172.

[3] 王彦明, 奉国和, 薛 云. 近年来 Hadoop 国外研究综述 [J]. 计算机系统应用, 2013, 22(6):1-5.

[4] 杜 江,张 铮,张杰鑫,等. MapReduce 并行编程模型研究综述 [J]. 计算机科学, 2015, 42(6A):537-541.

[5] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters [J]. Communications of the ACM, 2008, 51

(1):107-113.

[6] Redis [EB/OL]. 2016-01-28. <http://redis.io>.

[7] How fast is Redis [EB/OL]. 2013-08-20. <http://redis.io/topics/benchmarks>.

[8] Redis cluster specification [EB/OL]. 2014-10-09. <http://redis.io/topics/cluster-spec>.

[9] Rai P, Lal A. Google PageRank algorithm: Markov chain model and hidden Markov model [J]. International Journal of Computer Applications, 2016, 138(9):9-13.

[10] 李远方,邓世昆,闻玉彪,等. Hadoop-MapReduce 下的 PageRank 矩阵分块算法 [J]. 计算机技术与发展, 2011, 21(8):6-9.

[11] Och F J, Jahr M E, Thayer I E. Minimum error rate training with a large number of features for machine learning: US, 2013/0144593 A1 [P]. 2013-06-06.

[12] 胡爱娜,蔡晓艳. 基于 MapReduce 的分布式期望最大化算法 [J]. 科学技术与工程, 2013, 13(16):4603-4606.

[13] 曾超宇,李金香. Redis 在高速缓存系统中的应用 [J]. 微型机与应用, 2013, 32(12):11-13.

[14] Leskovec J, Rajaraman A. Mining of massive datasets [M]. Cambridge: Cambridge University Press, 2014.