信息科学系
低年级讨论班

（2）

裘宗燕

北京大学数学学院信息科学系

2010年3月～6月

MapReduce分布式编程

《代码之美》第23章

概要

- 本章描述 **Google** 的 **Map-Reduce** 的设计和实现
- **MapReduce** 是一种用于解决大规模数据处理的编程模型，开发的初衷是为了简化 **Google** 大规模计算程序的开发工作量
- **MapReduce** 程序能在大量廉价计算机构成的集群上自动并行化和执行。在此过程中，需要特别关注的问题包括：
  - 输入数据的划分
  - 集群上程序执行的调度
  - 机器错误的处置
  - 机器间通讯的管理
- 目标是使毫无并行和分布式系统设计经验的程序员也能很好地利用大型分布式系统里的资源（存储，通讯，计算能力等）

示例

- 设有200亿个文档，要统计每个独立单词在这些文档里出现的次数
  - 如果每个文档有20K，总数据量为400T（1T=1024G）
  - 一台计算机读入这些数据大约需要4个月
- 如果不怕等的时间长，机器的内存足够大，程序很容易写：

示例 23-1 朴素非并行的单词统计程序

```
map<string, int> word_count;
for each document d {
   for each word w in d {
     word_count[w]++;
   }
}
... 把统计数据保存到持久性存储器 ...
// 第一行定义了一个计数表格（通常为一个hash表）
// 循环中遇到一个单词，就将对应的计数值加一
```

示例

- 加速计算的一个方法是并行地对每个文档做同样工作

示例 **23-2**，并行化的单词计数程序

```
Mutex lock;  // 用一个锁保护 word_count
map<string, int> word_count;
for each document d in parallel {
  for each word w in d {
    lock.Lock( );
    word_count[w]++;
    lock.Unlock( );
  }
}
 ... 把统计数据保存到持久性存储器 ...
```

- 文档输入工作可以很好地并行化
- 实际中启动线程有些麻烦，这里用伪代码，忽略了一些复杂描述
- 主要问题是共享的数据结构 **word_count**。用一个数据结构可能造成严重的争用，成为性能瓶颈。可能改进是把数据结构分为很多桶（**buckets**），每个桶用一把锁，根据单词找锁和桶，可缓解这个问题（代码在下页）

示例

示例 **23-3**，使用分块存储的并行单词计数程序

```
struct CountTable {
  Mutex lock;
  map<string, int> word_count;
};
const int kNumBuckets = 256;
CountTable tables[kNumBuckets];
for each document d in parallel {
  for each word w in d {
    int bucket = hash(w) % kNumBuckets;
    tables[bucket].lock.Lock( );
    tables[bucket].word_count[w]++;
    tables[bucket].lock.Unlock( );
  }
}
for (int b = 0; b < kNumBuckets; b++) {
  ... save tables[b].word_count to persistent storage ...
}
```

- 程序仍然很简单
- 伸缩性（**scalability**）不超过一台机器上的处理器个数（很少几个）
- 完成工作需要几周时间
- 还没有考虑输入数据的存放（几百**TB**，内存？）
- 进一步考虑伸缩性，需要把计算分配到多台计算机。先假定计算机不出故障，可以在网络化的计算机集群上启动很多并行进程

- 这里用**256**个桶（和**256**个锁），根据单词的**hash**值决定投入哪个桶

示例

1000个输入进程，256个输出进程

D是分配给每个输入进程的文档数

开始时启动所有输入和输出进程

一个输入进程里有一组统计表，对应每个输出进程有一个统计表。统计完成后将给各输出进程的表打包送去

输出进程取得各输入进程送来的包，在自己的统计表里综合得到的结果

示例 23-4，并行单词计数程序，使用划分的进程

```
const int M = 1000;   // Number of input processes
 const int R = 256;    // Number of output processes
 main( ) {
   // Compute the number of documents to assign to each process
   const int D = number of documents / M;
   for (int i = 0; i < M; i++) {  fork InputProcess(i * D, (i + 1) * D);  }
   for (int i = 0; i < R; i++) {  fork OutputProcess(i);  }
   ... wait for all processes to finish ...
 }

 void InputProcess(int start_doc, int end_doc) {
   map<string, int> word_count[R];     // Separate table per output process
   for each doc d in range [start_doc .. end_doc-1] do {
     for each word w in d {  int b = hash(w) % R;  word_count[b][w]++;    }
   }
   for (int b = 0; b < R; b++) {
     string s = EncodeTable(word_count[b]);   ... send s to output process b ...
   }
 }

 void OutputProcess(int bucket) {
   map<string, int> word_count;
   for each input process p {
     string s = ... read message from p ...
     map<string, int> partial = DecodeTable(s);
     for each <word, count> in partial do {   word_count[word] += count;   }
   }
   ... save word_count to persistent storage ...
 }
```

示例

- 最后这个方法可以很好应用在工作站网络上，但
  - 它比较复杂，难以理解
  - 这里隐藏了打包和解包的细节
  - 仍没有看清楚启动和同步不同进程的细节，
  - 没有很好的容错方法
- 处理出故障的机器，可以考虑扩展**23-4**的代码
  - 如果某台机器没完成工作，就重做相应工作
  - 还要避免重复计数。一个可能方法是给数据加上分代编号（**generation number**），处理程序检查分代编号，避免重复
  - 总之处理故障很不容易

MapReduce编程模型

- 对比示例 **23-1** 和 **23-4**，可以看到在 **23-4** 里简单的单词计数工作被掩盖在大量并行计算管理的细节中
- 如果能把处理要解决问题的计算和并行化细节分开，就可能得到一种很好的通用并行库或者编程系统。这种系统不仅适用于做单词计数，还可能适用于完成其他大规模数据处理工作
- 下面要使用的并行模式是
  - 对每个输入记录，提取出我们关心的**key/value**对
  - 对提取到的每个**key/value**对，将其与其他有同样**key**的值合并（进程中可能牵涉到对值的过滤，累积或变换）
- 重写具体工作的特定应用逻辑，实现上述两种功能模式
  - 用两个函数分别做一个文档的单词计数和文档计数归并
  - 一个称为映射（**map**），另一个称为化简（**reduce**）

MapReduce编程模型

Map处理一个文档，对其中每个单词送出一个中间结果

Reduce接到一个单词和一些值（计数），归并计数（对其求和）后送出

这是两个基本操作，具体使用看调用它们的驱动程序和低层**EmitIntermediate**和**Emit**的实现

应看到：

这里**Map**和**Reduce**实现的是实际应用逻辑，可以用不同的控制逻辑驱动其执行

示例 **23-5**，把单词计数问题分解为**Map**和**Reduce**

```
void Map(string document) {
  for each word w in document {
    EmitIntermediate(w, "1");
  }
}

void Reduce(string word, list<string> values) {
  int count = 0;
  for each v in values {
    count += StringToInt(v);
  }
  Emit(word, IntToString(count));
}
```

MapReduce

一个简单的单机版驱动程序，调用上面的函数完成单词计数

Driver 的工作：

对每个文件调用 **Map** 一次，其中调用底层函数把数据填入数据结构

然后调用 **Reduce** 把数据汇总输出

EmitIntermediate 给每个单词的数据表加一项

这个驱动程序意思不大，只是说明一下可能如何使用 **Map** 和 **Reduce**

示例 **Example 23-6**
Driver for Map and Reduce

```
map<string, list<string> > intermediate_data;

void EmitIntermediate(string key, string value) {
  intermediate_data[key].append(value);
}

void Emit(string key, string value) {
  ... write key/value to final data file ...
}

void Driver(MapFunction mapper,
                ReduceFunction reducer) {
  for each input item do {    mapper(item);   }
  for each key k in intermediate_data {
    reducer(k, intermediate_data[k]);
  }
}

main( ) {   Driver(Map, Reduce);  }
```

其他MapReduce编程实例

进一步探讨更有趣的**Driver**实现之前，先看另一些可能如何通过 **Map-Reduce** 模型解决的问题

- 分布式 **grep (distributed grep)**

  grep：找出与给定正则表达式匹配的正文行

  Map 将匹配的行发出，**Reduce** 直接把结果拷贝到输出

- 反向 **Web** 链接图 **(reverse web-link graph)**

  如果**Web**上**url1**的网页里有**url2**链接，图中记一条**url2**到**url1**的边

  Map 对 **Web** 文档中每个链接送出一个 **<target, source>** 对

  Reduce 把与同一个 **target url** 相关的 **sourse url** 汇集为一个表，送出一个对 **<target, source**的表**>**

- 主机的词汇向量 **(term vector per host)**

  一个或一组文档里出现的最关键词汇及其出现频率**<term, freq>**的列表。**Map**从每个输入文档得到**<host, termvector>**（主机名取自**url**），**Reduce**汇集这种对，合并词汇向量，删除非频繁词，得到新的**<host, termvector>**

其他MapReduce编程实例

- 反向索引（**Inverted index**）

  单词到包含单词的文档列表的映射（文档通常用用数字**ID**表示）。**Map**解析文档，输出一个**<word, docID>**表；**Reduce**接受一个单词的序对，输出序对**<word, docID**表**>**。很容易扩充这一模式，增加单词在文档的定位信息

- 分布式排序

  针对某个关键词排序。**Map**从记录中提取关键词，送出**<key, record>**对；**Reduce**照样输出，排序依赖于后面介绍的划分工具和顺序性质

- 许多其他问题可以很容易用**Map-Reduce**计算的形式来表达。有时需要表达为一系列**Map-Reduce**步骤或**Map-Reduce**迭代过程，以一个**Map-Reduce**步骤的输出作为下一**Map-Reduce**步骤的输入
- 人们发现，按照**Map-Reduce**方式思考问题并不困难，从**2002**年开始做出了几个**Map-Reduce**计算过程，到**2006**年底人们已经写出超过**6000**个**Map-Reduce**过程，这些过程由超过**1000**个不同开发者开发，绝大多数原来都没写过并发程序和分布式程序

一个分布式Map-Reduce实现

- **Map-Reduce**模型的主要优点是应用逻辑与并行/故障处理等底层细节分离。不同平台可采用不同实现方法，正确选择依赖于具体环境。如小型共享内存机器用一种实现，大型**NUMA**（非一致访问架构）多处理器系统用另一种实现，规模更大的网络计算机集群用另外一种实现
- 下面介绍一种更复杂的实现，基本上就是**Google**常用的计算环境的情况，适于运行大规模的**Map-Reduce**作业。所用环境是由大量普通计算机通过以太网交换互连构成的大型集群。环境里：
  - 机器为双**x86**处理器，**2-4G**内存，运行**Linux**
  - 机器通过普通网卡互连（通常**1G**以太网），一个机架**40-80**台机器，机架连接中心交换机，分配给每个机架的带宽**50-100Mb/s**
  - 存储介质是各机器上的普通**IDE**硬盘，用**GFS**分布式文件系统管理硬盘数据。**GFS**通过多拷贝的方式提高硬件的可靠性：文件划分为**64M**的块，每个块在不同机器上有**3**个拷贝
  - 用户把作业提交调度系统，每个作业包含一组任务，由调度系统映射到集群里的某一组机器上

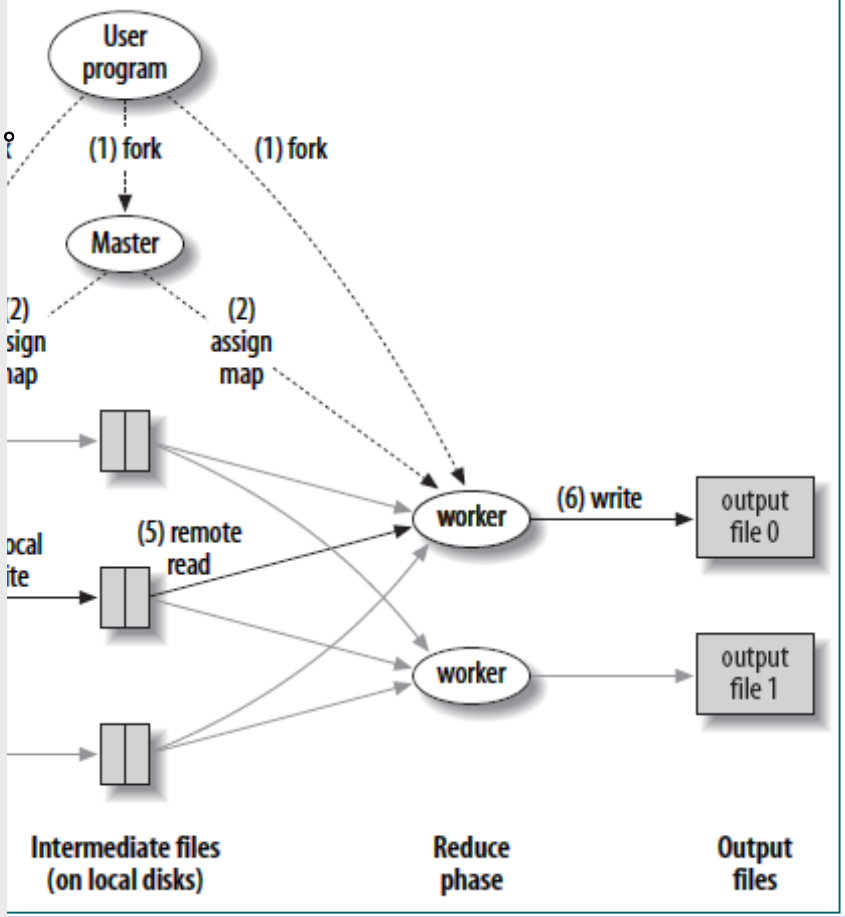
执行情况

- 输入数据划分为**m**块后将**Map**调用分配到多台机器。划分由其他机器并行处理
- 将中间结果通过**key**空间划分为**R**块（如用**hash(k)%R**）后调用**Reduce**

reduce工作站接到工作数据位置后用远程调用从**Map**工作站的硬盘读数据。读完后做**reduce**按**key**排序使**key**相同的数据汇集到一起。中间数据太大时用外存排序

reduce工作站将**k/v**对送给用户**reduce**函数，把生成的**k/v**对加入输出文件。**reduce**工作站完成时通知**mastor**，要求另一**reduce**任务

所有**map/reduce**任务都完成时把控制交回用户程序，产生的输出就是**R**个**reduce**文件



MapReduce划分输入为**16-64M**的块后向调度器请求启动程序的多个拷贝

有一个特殊拷贝**master**，它将其余任务分配给**M**个**map**工作站和**R**个**reduce**工作站（**worker**）

map工作站读输入生成**k/v**对缓存在内存，周期性地写入本地硬盘，用划分功能划分为**R**个桶

一个**map**工作站完成时通知**master**，**master**将其生成的数据传给**reduce**工作站。如还有**map**工作就再分它一个

执行情况：细节

一些实现细节使上述实现可以在目标环境中取得很好的性能：

- 负载平衡：一个**MapReduce**作业的任务数通常多于可用工作站，因此**master**会给一个工作站分配多个任务。完成任务的工作进程会得到新任务，速度快的机器分到更多任务。即使机器性能差别很大，也能基本达到负载平衡
- 容错： **MapReduce**必须处理出错。**master**保留各工作站完成**map/reduce**工作的状态，周期性地**ping**工作站，连续几次**ping**没接到回复就宣告该工作站死亡，并将其工作重发给另外的空闲工作站。典型**MapReduce**执行系统的工作站数是作业的**50**倍，恢复操作很快。**master**把调度状态写入一个持久性日志。如**master**死亡，集群调度重启一个**master**，让它根据日志重建工作状态
- 本地性：输入数据由**GFS**管理，保存在执行**map**的机器或同一机架的硬盘里。特定作业的**master**找到输入数据的位置（可能有多份数据），然后在靠近数据的机器上建立工作站。因此，多数工作站是在主机的硬盘里读数据
- 任务备份：常会有个别拖后腿的工作进程，其任务很长时间不能完成（任务工作量大或机器慢）。通过任务备份可以缓解这种问题。当只剩少量任务时，为这些任务另行分配空闲工作站同时做。一个工作站完成就是任务完成

模型扩展

使用**MapReduce**，多数情况只需写出**Map**和**Reduce**函数。扩充基本模型是希望提供一些实际中有用的功能

- 划分功能：用户给定说明希望的**reduce**任务数**R** ，系统自动按中间关键码划分中间数据。默认划分函数是 **hash(key) % R**，这自动平衡了**R**组数据

  允许用户提供划分函数，例如用户可能希望同一主机的条目划分在一起

- 顺序保证： **MapReduce**对中间数据排序，**key**相同的中间结果集合在一起。多数用户觉得在按**key**排序的数据上**reduce**更方便，这时可用这项功能

- 跳过坏记录：用户代码的**bug**可能导致**Map**和**Reduce**函数在某些数据上必定崩溃，使**MapReduce**在做了巨量工作后失败。这时首先应考虑排除**bug**，但有时不可行（例如用第三方库，没有源码）。为此提供一种可选执行模式：让**MapReduce**库检测导致崩溃的记录，遇到时跳过，以保证可以继续工作

  每个工作站安装了一个信号处理程序（**signal handler**），捕捉段违规和总线错。调用用户**Map/Reduce**操作前， **MapReduce**库用一个全局变量保存记录的序列号。如果用户代码产生出错信号，其信号处理器就给**master**发一个包含序列号的**last gasp**（临终遗言）的**UDP**包，说明如果再次启动这个**Map**或**Reduce**工作时应要求工作站跳过这个记录

总结

- 事实证明**MapReduce**是对**Google**非常有用的工具
- 至**2007**年初，已开发了超过**6000**个按**MapReduce**模型编写的程序
- **Google**每天运行约**35000**个**MapReduce**作业，处理大约**8P**输入数据（大约每秒**100GB**）
- 最初写**MapReduce**是想作为重新开发分搜索引擎系统的一部分，后来发现这种模型在许多地方都很有用
  - 机器学习
  - 统计机器翻译
  - 日志分析
  - 信息检索试验
  - 通用的海量信息处理和海量计算
- 书上给出了一些参考文献，有兴趣可以找来读读

C++里单词计数的完整代码

```
// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value(  );
    const int n = text.size(  );
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;

      if (start < i)
        EmitIntermediate(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);
```

```
// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done(  )) {
      value += StringToInt(input->value(  ));
      input->NextValue(  );
    }

    // Emit sum for input->key(  )
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  MapReduceSpecification spec;
  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input(  );
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }
```

```
  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100   ...
  MapReduceOutput* out = spec.output(  );
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2,000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort(  );

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.
  return 0;
}
```

研究问题

这一工作给我们的提示有两个方面。由此带来两方面的研究问题：

1. 计算的描述模型。这方面研究永远是最重要的
   - 新问题类
   - 新的计算环境
2. 使用**MapReduce**模型
   - 把其他实际的有代表性的大规模计算问题归结到**Map-Reduce**

讨论