大规模图计算模型——Pregel

■ 文 / 肖康

图是计算机科学中的经典问题，在现实中也有很多应用与图相关，例如Web链接组成的图、SNS中人与人之间的各种关系图、电子地图中的路径搜索等。当图的规模很大时就需要一种有效的方式来处理与图相关的计算，有资料显示在Google大规模数据处理80%是用MapReduce，而其余20%是用另一个面向图计算的模型——Pregel。Google 2010年发表在SIGMOD的论文“Pregel: a system for large-scale graph processing”对这一模型进行了揭秘，论文中提到已经有大量的生产应用通过Pregel开发运行，而且很少有不能在Pregel上实现的实际图计算应用。

目前针对图计算的解决方案中，单机图算法库比较成熟但扩展性不好，很难处理Web规模的图计算；并行图算法库又缺少容错，在大规模数据处理中出错又是很正常的；基于MapReduce的多轮迭代又不够高效，因为需要反复存储和读取迭代的结果；MPI实现并行计算比较灵活，但需要应用考虑通信、同步、容错等，因此编程比较复杂。

与以上计算模型相比，Pregel具有以下几个特点和优势：第一，原生的图计算API，简单易用且表达力强；第二，高效；第三，很好的容错和可靠性；第四，可扩展性好。

模型

Pregel计算模型可以概括为BSP（Bulk Synchronization Parallel）+ 节点为中心的图计算API + 容错。

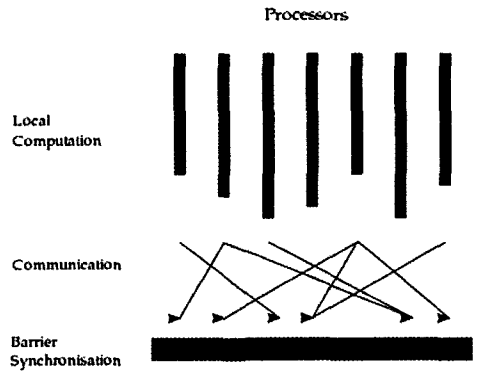
如图1所示，经典的BSP模型计算由多次迭代组成，每次迭代称为一个superstep，每个superstep分为以下3个步骤。

计算：各个进程独立同步进行，且每个superstep执行同样的逻辑。

通信：各个进程之间进行通信，通信的数据在下一个superstep作为输入使用。

同步：所有进程进行同步，然后从1开始下一个superstep。

图1 经典的BSP模型



Pergel提供了如图2所示的以节点为中心的API，它具有以下几个特点。

■ 以节点为中心，所有的边是源节点的附属品。一个节点的数据包括三项：节点的值、它的出边的值、下一个superstep要处理的消息，在API中分别由VertexValue、EdgeValue、MessageValue类型表示。

■ Compute()方法是计算的核心。可以进行的操作包括：读取上一个superstep发送到该节点的消息，向其他节点发送消息，读取和修改节点的值，修改出边的值，修改图的拓扑。

■ 所有节点初始为Active状态，节点在Compute()中调用VoteToHalt()进入InActive状态，当有消息发送到InActive状态的节点时，它又被激活到Active状态；如果所有节点都进入InActive状态且没有任何消息，则整个计算结束。

图2 Pergel提供的以节点为中心的API

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

与BSP对应，Pregel计算模型的superstep也分为下面三个步骤。

计算：对于图的每个节点Pregel执行应用提供的Compute()方法，每个节点独立并行地进行计算。

通信：通信只能通过收发消息进行。节点之间通信是和计算并发进行的，计算过程中可以向图中的其他节点发送消息，这些消息的收发由Pregel自动完成而不需要应用关心，当前superstep发送的消息作为下一个superstep的输入消息，而在当前superstep不可见。Pregel不保证发送到一个节点的消息的顺序，但是保证一定会发送到且不会重复。节点一般向其相邻节点发送消息，也可以向其他任何节点发送消息。

同步：当一个superstep的所有节点计算结束且所有消息都已经收发完成，就可以开始下一个superstep。

Pregel提供了自动容错的机制，当部分节点的计算由于各种原因如机器宕机、网络异常等失败后，会通过自动重算部分superstep进行恢复，整个计算时间可能会拉长但还是会完成，并不会因为系统异常而失败。

如图3所示，一个典型的Pregel计算由输入、supersteps、输出组成。输入是要处理的图，会按照节点进行划分，输出是最终各个节点的值，多数情况下也是一个和输入类似的图，supersteps就是上面描述的计算过程。

实现

总体架构

Pregel的实现从架构上是master/slave的结构，如图4所示。输入的所有节点分割成一个个partition，master负责将每个partition分配给各个worker，协调各个worker执行superstep，进行全局信息汇总和状态显示；worker负责保存节点的状态，执行superstep。

图4 Pregel的架构

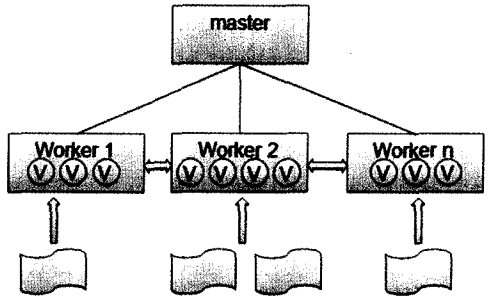
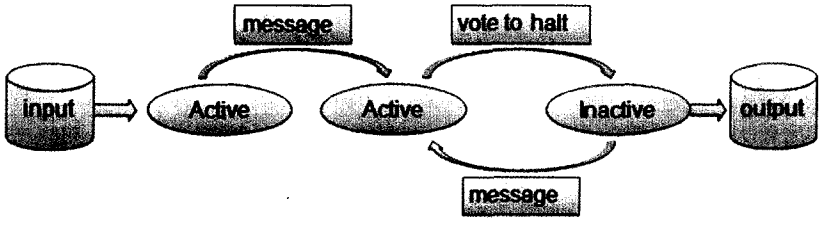


图3 一个典型的Pregel计算的组成



master/worker

master负责将partition分配给各个worker，这里允许应用指定分配的策略，因为不同的分配方式可能对计算性能产生影响。分配的单位是partition而不是节点，因此单master就可以处理很大的图。一个worker可能会分配到多个partition，这样并发度更高且有利于worker之间负载均衡。

master对spuerstep进行同步，一个supserstep的所有节点计算和消息收发都完成后，便通知worker开始下一个superstep。在一个superstep中，worker循环执行它负责的所有节点的Compute()方法，每个线程处理一个partition。

Pregel具有以下几个特点和优势：第一，原生的图计算API，简单易用且表达力强；第二，高效；第三，很好的容错和可靠性；第四，可扩展性好。

worker在内存中维护每个节点的数据，包括节点的值、出边、接收的消息、Active/InActive状态，计算的进程一直运行而不是每个superstep启动新的进程，这点和基于MapReduce的多轮迭代有很大区别，减少了每次迭代启动进程和加载、保存数据的开销。

对于每个节点接收的消息，worker维护两个队列，一个是上一个superstep收到的消息，一个是当前superstep正在接收的消息。这样每个节点在计算过程中使用上个superstep消息的同时，可以异步接收当前superstep发送给它的消息，使计算和通信并行起来。Compute()中发出的消息，如果是发往本机节点的就直接放入其消息队列，如果是发往其他机器worker则先缓存起来，积累到一定量才异步地发送出去，以提高网络传输的效率。

Combiner和Aggregator

节点规模比较大时，发送的消息量可能比较多，Pregel提供了Combiner机制减少消息量，应用可以提供一个Combiner，消息发送端和接收端可合并一些消息，以减少发送消息需要的网络带宽和存储消息的内存资源。这一点和MapReduce中的Combiner比较类似。由于Combiner需要满足交换律和结合律（比如累加计数），所以无法提供默认实现，只能由应用提供。

Pregel还提供了一个和MapReduce Counters类似的Aggregators机制，用来汇聚全局信息。Worker维护每个节点的Aggregators，本地汇聚后上报给master进行全局汇聚，最后把全局汇聚的值发给各个worker。Aggregators和Counters有两点不同：一方面Aggregators运行提供汇聚函数（如取最大值），而Counters只是累加；另一方面全局汇聚值还会发给worker在下一个superstep使用。

容错机制

Pregel通过checkpoint来实现容错：master定期向slave发ping，woker收到ping后向master回应；worker如果长时间未收到ping就自动退出；master如果长时间未收到回应就认为worker出错。superstep开始之前，worker把节点数据（包括节点、出边，收到的消息）保存到持久化存储如GFS中，如果master检测到worker出错，会重新将所有partition分配给正常的worker，让每个worker从持久化存储中加载对应partition最近一次保存的节点状态，然后从这个superstep开始执行。

即使上面的容错过程中没有出错的worker节点，也要从上次checkpoint的superstep重新执行，因此Pregel正在开发一种更高效的方式（Cofined Recovery）只重新执行出错的supersteps。这种方式需要worker记录上次checkpoint开始所有发出的消息，当某个worker出错时，在其他worker上加载对应节点的checkpoint，其他正常的worker把记录的消息重新发送给这些节点，这些节点就能从上次checkpoint恢复到worker出错时的superstep，而其他worker上的节点不需要恢复。这种方式的优点是只用恢复出错的worker，可以减少重算所有节点带来的浪费，因此恢复的时间也能缩短，而缺点是需要记录更多的数据而且过程相

对复杂一些。

应用

这里以论文中单源最短路径的实现说明Pregel的应用。求单源最短路径是找出图中某个点到其他所有点的最短距离，在Pregel中实现如下（图5）。

图5 单源最短路径的实现

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

■ 点、边、消息的类型都是整数，边的值是两个相邻点的距离，初始值由输入图决定。点的值是源点到该点的最短距离，源点初始值为0，其他点为无穷大。

■ 每次superstep中，节点收到来自相邻节点的消息，消息的值是源节点到相邻节点的最短距离。如果消息的最小值比该节点本身的值小，说明源节点通过某个相邻节点到该节点的距离比当前最短距离还小，就将自己的值替换为最小值，且将自己的值加上出边值发送给出边的目的节点。

■ Superstep一直到没有节点向外发消息，也就是每个节点的最短距离就是当前节点值时结束。

根据论文中的测试数据，Pregel的扩展性还是相当不错的，worker数不变时对于SSSP图的节点数增加计算时间随之线性增长，而图的节点数不变时worker数增加16倍使性能提高10倍。

Pregel在Google的使用过程中不断得到发展，比如加入Aggregators、Web展示页面、单元测试和单机调试工具等，这些经验也是设计其他计算模型和系统时值得借鉴的，因为易用性的改进对于一个模型和系统的广泛使用也是很关键的。Pregel目前处理上亿节点规模的图已经不是问题，为了能处理更大规模的图，还会有一些优化和改进：内存不够用时worker利用本地磁盘存储一些数据；动态重新分配partiton到不同的worker以获得更优的性能；简化BSP的同步模型，使得那些运行得快的worker不用等待少数慢的worker等。

总结

Pregel的优势在于简单、高效、容错、可扩展，它的计算模型和MapReduce一样比较简单、容易理解，作为一个提供了自动通信、容错等机制的分布式计算模型，它的API比较友好，使应用专注于单个节点的本地操作而不是通信、容错等分布式相关的问题。虽然没有MPI灵活，但它提供的自动消息收发、容错等机制对应用来说很有价值。在扩展性方面，与相关图计算的库和框架相比能扩展到更多的机器。

当然，Pregel不如MapReduce应用广泛，可能也有一些原因，比如：模型和API专注于多轮迭代的图计算，有些计算可能不方便向这个模型映射；对于稠密图和需要大量通信的计算，还是不太高效等。

不管怎样，作为一个新的计算模型Pregel是值得研究和使用的，目前已经有一些相关的开源实现，例如Apache基于Hadoop的BSP模型Hama，Github上多个Pregel项目，RAVEL基于Hadoop的Pregel实现GoldenOrb。虽然都还没有像Hadoop这样成熟的产品，还是值得关注和贡献的。

肖康

百度基础架构部高级工程师，百度分布式计算方向核心开发人员，百度下一代计算系统模块负责人。曾受邀在Hadoop in China 2010会议上进行技术讲座。