SHANGHAIJIAOTONGUNIVERSITY

计算机体系结构

实验报告

学生姓名：林江浩

学生学号：517021910674

专　　业：软件工程

指导教师：李　超

学 院(系)：电子信息与电气工程学院

目录

1. 实验环境.....1
2. 算法的描述与实现.....1
    2.1 Pregel.....1
    2.2 SSSP（Single Source Shortest Path）.....2
    2.3 PageRank.....3
3. 实验过程与分析.....3
    3.1 SSSP 算法的实验分析.....3
    3.2 PageRank 算法的实验分析.....5
    3.3 实验的数据总结.....6
4. 总结.....6

1. 实验环境

- 操作系统：Windows 10
- CPU：Intel(R) Core™ i7-6700HQ CPU @ 2.60GHz
- 内存：8GB
- JDK：1.8
- Hadoop：2.7.0
- Spark：2.4.5
- Scala：2.11.12

2. 算法的描述与实现

在这一部分，我会对本次实验中用到的 Pregel 计算框架和两个算法（SSSP、PageRank）进行描述，并对我的实现代码进行部分讲解。

2.1 Pregel

Pregel 是一个基于 BSP 的大规模分布式图计算框架。主要用于图遍历（BFS）、最短路径（SSSP）、PageRank 计算等计算。

在 Pregel 计算模式中，输入是一个有向图，该有向图的每一个顶点都有一个相应的独一无二的顶点 ID （Vertex Identifier）。每一个顶点都有一些属性，这些属性可以被修改，其初始值由用户定义。每一条有向边都和其源顶点关联，并且也拥有一些用户定义的属性和值，并同时还记录了其目的顶点的 ID。

在 Pregel 中，顶点有两种状态：活跃状态（Active）和不活跃状态（Halt）。如果某一个顶点接收到了消息并且需要执行计算那么它就会将自己设置为活跃状态。如果没有接收到消息或者接收到消息，但是发现自己不需要进行计算，那么就会将自己设置为不活跃状态。这种机制的描述如下图：

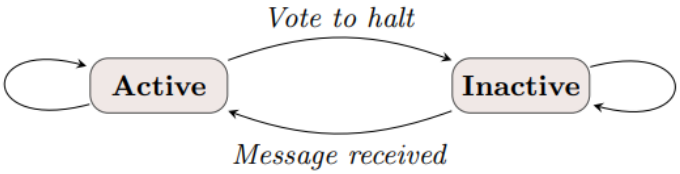

图 1.1.1 Pregel 顶点的状态机

Pregel 中的计算可以分为一个个超步（Superstep）的不断迭代，超步的执行流程如下：

(1) 首先读入图数据并进行初始化。

(2) 将每个顶点的状态均设为活跃状态，每个顶点根据预先定义好的 SendMessage 函数向周围的顶点发送信息。

(3) 每个顶点接收信息后，如果发现需要计算，则根据预先定义好的 Compute 函数对接收到的信息进行处理，这个过程可能会更新自己的信息。如果接收到信息但是不需要计算，则将自己的状态设置为不活跃状态。

(4) 所有的活跃顶点按照 SendMessage 函数向其他顶点发送新的消息，一个超步结束。

(5) 下一个超步开始，重复步骤 3 和步骤 4，直至所有的顶点均变为不活跃状态，则整个计算过程层结束。

2.2 SSSP（Single Source Shortest Path）

按照 1.1 中的描述，首先我们要从对应的 txt 文件中读取图数据后进行初始化，并用每个顶点的属性值来表示源点到该顶点的最短路径，即仅有源点到源点的距离被初始化为 0.0，对所有的非源顶点，我们都将其属性值设置为无穷，之后的计算中如果遇到更短的路径，则替换即可。若源点到某顶点不可达，则对应路径长度为无穷。

```
val graph = GraphLoader.edgeListFile(sc, path = "C:\\Users\\22859\\Desktop\\web-Google.txt")
val sourceId: VertexId = 0 // Set the source vertex
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.PositiveInfinity)
```

图 1.2.1 SSSP 初始化图数据的代码

而接下来基于 Pregel 的 SSSP 算法，我将整个计算过程分为三个部分，对应代码中调用 Pregel API 时提供的三个参数：计算、消息发送和消息合并。具体代码可见下图：

```
val sssp = initialGraph.pregel(Double.PositiveInfinity) (
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => { // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a, b) => math.min(a, b) // Merge Message
)
```

图 1.2.2 基于 Pregel 的 SSSP 的核心代码

当一个顶点接收到来自其他顶点的消息后，就会将自己的状态设置为活跃，并且进行计算，计算过程就是取最小值即可，因为任务目标是计算源点到该顶点的最短路径。

每一个超步的末尾，所有的活跃顶点都会向其他节点发送消息，对于每一个 triplet，我们会比较 triplet.srcAttr + triplet.attr 和 triplet.dstAttr 的大小关系，若前者小于后者，则说明源点到 triplet.dstID 的路径长度可以被缩短，因此向 triplet.dstID 发送消息。

在每一个超步的末尾，很多顶点都会收到若干条不同的消息，若依次进行处理会造成较大的开销。而因为 SSSP 的目标是求出源点到该顶点的最短距离，因此我们可以将所有信息合并，保持所有消息中的最短举例即可。

如此一来，整个计算过程结束（即所有顶点都被设置为不活跃状态）的时候，就意味着所有顶点的属性值（源点到该顶点的最短路径）不再被更新，也就是完成了单源点到所有顶点的最短路径的计算。

2. 3 PageRank

因为 GraphX 库中内置了 PageRank 的接口，因此我们不需要像 1.2 的 SSSP 一样自己去实现对应的计算函数、消息发送函数等，只需读入图数据后，调用对应的接口即可。

```
println("==>Run PageRank")
val ranks = graph.pageRank( tol = 0.0001).vertices
```

图 1.3.1 调用 GraphX 的 PageRank 的接口代码

通过阅读 GraphX 中 PageRank 相关的源码可以发现，内置的 PageRank 算法提供了两种不同的 PageRank 实现，分别是静态算法和动态算法。

对于静态算法，我们会在调用 PageRank 接口的时候传入一个整数参数 number，当 PageRank 的迭代次数达到 number 后计算立即停止，不论整体收敛与否。对于动态算法，我们会在调用的时候提供一个 tol 参数，当前后两次迭代的结果值小于 tol 时，则对应顶点被设置为不活跃状态。当且仅当所有顶点的更新差值都小于 tol 时（即所有顶点都被设置为不活跃状态），整体的计算才会结束，和 Pregel 的计算框架相吻合。以你本次实验中，我调用得到是动态算法下的 PageRank 接口。

3. 实验过程与分析

本次实验，我利用 Windows 自带的性能检测工具 Perfmon，针对 Wikipedia 和 Google 两个数据集、SSSP 和 PageRank 两个算法进行测试，测试指标包括内存利用率、CPU 使用率、L2 缓存缺失率、L3 缓存缺失率。

为了让 Perfmon 能够检测到缓存状态，我通过编译 PCM 工具向 Perfmon 添加了相应的服务，PCM 开源地址如下：https://github.com/opcm/pcm。但是因为 PCM 工具的原因，安装服务后的 Perform 仍然只能检测 L2 和 L3 缓存的缺失率，Windows 下 L1 缓存缺失率的检测目前没有找到合适的工具。

因为两个数据集在同一算法上的指标测试结果具有共性，因此接下来根据算法进行分类实验分析，并从中穿插对两个数据集的对比讨论，而算法之间的对比分析，我穿插在 3.2 节 PageRank 算法的实验分析中。

3. 1 SSSP 算法的实验分析

内存使用率和 CPU 使用率测试记录结果如下：

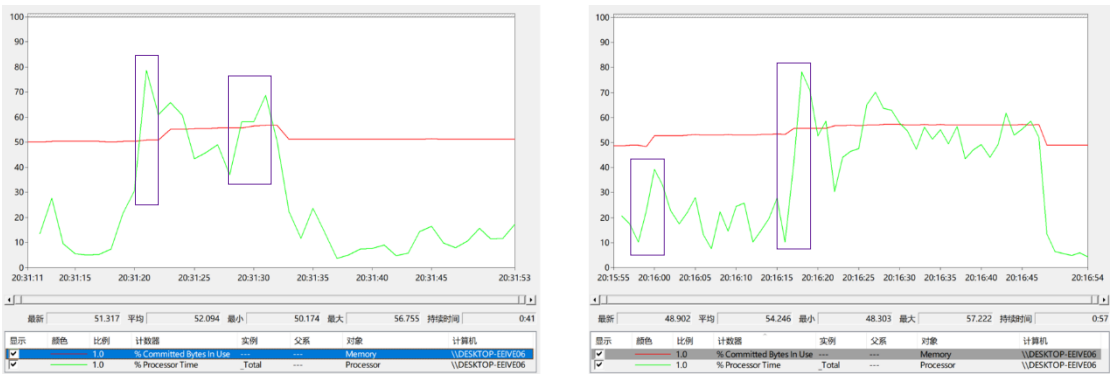

图 3.1.1 SSSP 算法的内存和 CPU 使用率（左为 Wiki，右为 Google）

可以发现内存占用率（红色）经历了四个阶段，分别是显著上升、缓慢爬坡、趋于平稳、显著下降。而 CPU 占用率（绿色）有两次拔升过程（可见紫色方框），分别与内存占用率的显著上升阶段和趋于平稳阶段相对应。

内存占用率显著上升阶段，是程序 spark 运行环境的构建过程，因为我显示扩大了 JVM 的内存和堆的大小，故有一个较为明显的内存占用上升，同时 CPU 占用率有所上升。缓慢爬坡阶段，是程序开始读入图数据并构建图数据结构的过程。趋于平稳阶段，是程序开始图计算过程，这时不用再去大量占用额外内存资源，但是需要进一步占用 CPU 资源，因此 CPU 占用率在这个阶段有显著提高。显著下降阶段，说明程序执行完毕，归还内存和 CPU 资源。

而对比来看，因为 Google 数据集比 Wikipedia 数据集要大，因此 SSSP 算法在 Google 数据集上的运行时间要更长（缓慢爬坡阶段+趋于平稳阶段），内存占用率也略高。

L2 和 L3 缓存缺失率测试结果如下：

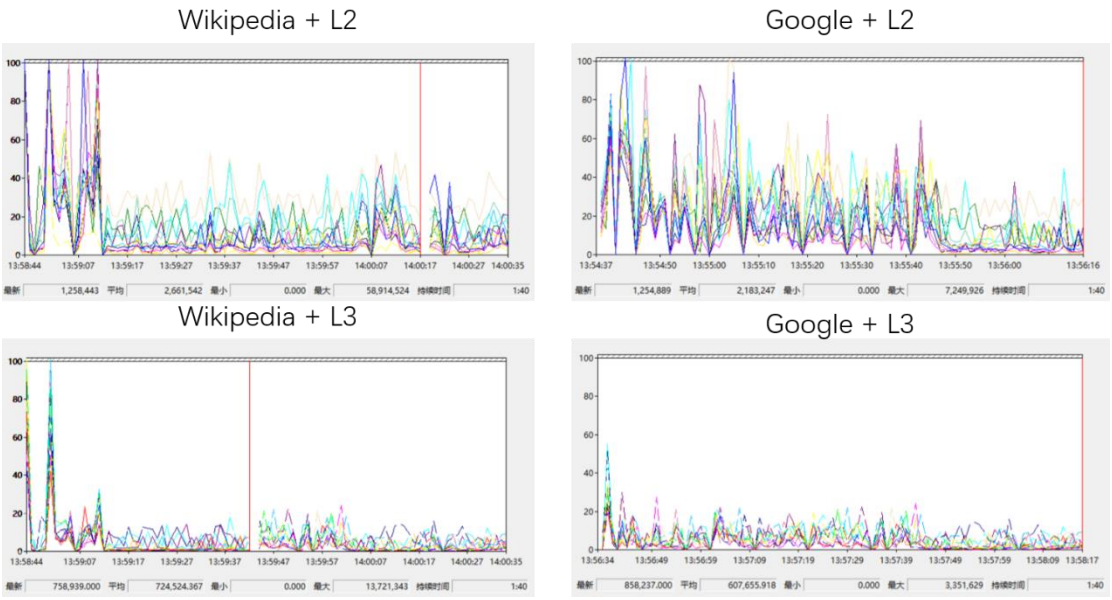

图 3.1.2 SSSP 算法的 L2、L3 缓存缺失率

整体而言，四幅图都呈现出的规律是：

(a) L2 和 L3 缓存的整体缺失率较高，尤其是前期运行环境和图数据的构建过程，之后的计算过程相对降低，在程序结束后回落至较低水平。这一点契合了图数据加载与计算过程对数据访存高要求的特点。

(b) 同时缓存缺失率均呈现周期性波动，可能是由缓存替换策略和 CPU 调度导致的。

横向对比两个实验数据集，因为 Google 数据集更大，因此 Google 数据集的缓存缺失率均略高于 Wikipedia 数据集。并且在 L2 缓存上，这个大小关系体现的更加明显。

纵向对比 L2 和 L3 缓存，因为 L2 缓存比 L3 缓存更高一级，也更小，因此对同样的数据集，L2 缓存的缺失率明显比 L3 要高，这一点符合预期。

3. 2 PageRank 算法的实验分析

内存使用率和 CPU 占用率测试记录结果如下：

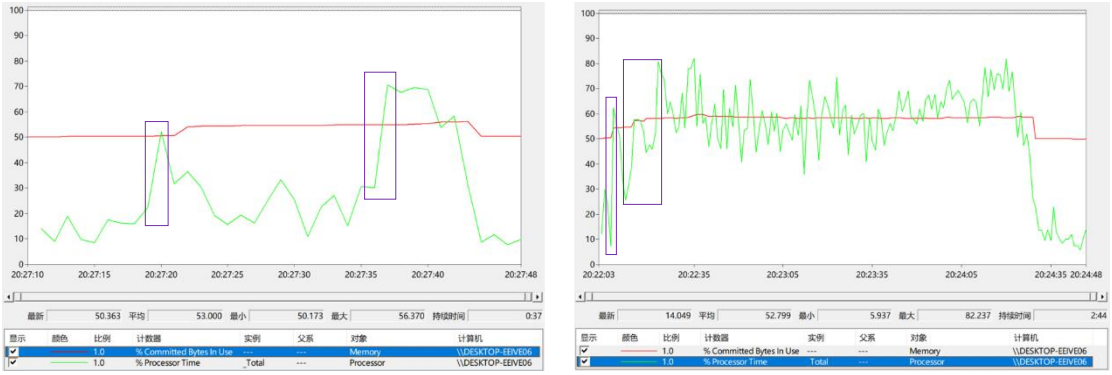

图 3.2.1 PageRank 内存和 CPU 占用率（左为 Wiki，右为 Google）

对于内存利用率，可以发现，和 SSSP 算法结果类似，内存占用率也是分为四个阶段：显著上升、缓慢爬坡、趋于平稳、显著下降，并且也是和程序运行的四个阶段一一对应，即程序构建 spark 运行环境、加载并构建图数据、图计算、程序运行结束。

对于 CPU 占用率，也是和 SSSP 算法结果类似，有两次抬升（见紫框）。Wikipedia 数据集的表现和 SSSP 算法几乎一样，都是在程序构建运行环境和图计算的时候有一个显著的 CPU 占用率提升。而 Google 图的 CPU 占用率也有两次显著提升，而之所以看起来时间相隔很近，是因为比例尺问题，PageRank 算法在 Google 图上的图计算花费了很多时间，因此在结果绘制上，前期的环境构建和加载数据的时间比例减小了。同样的，因为比例尺问题，Google 图在图计算阶段的 CPU 占用率的波动显得更剧烈。

对比来看，同样是因为 Google 图比 Wikipedia 图要大，因此占用相对更多的内存，并且图计算的耗时更长，且耗时的增长幅度明显高于 SSSP 算法，这应该与算法本身特性有关。

L2 和 L3 缓存缺失率的测试结果记录如下：

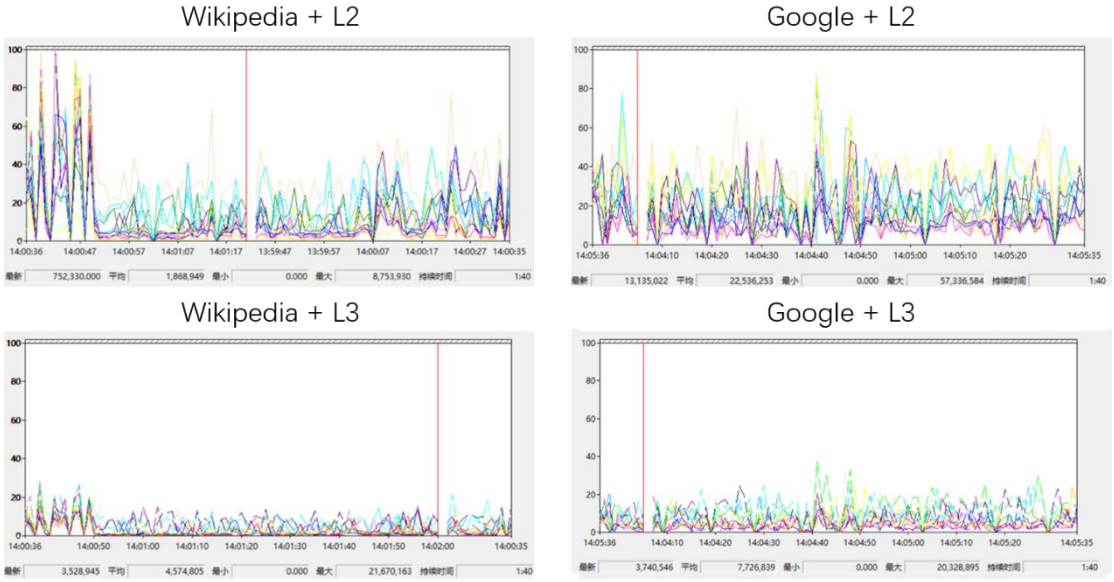

图 3.2.2 PageRank 算法的 L2、L3 缓存缺失率

在缓存缺失率的表现上，PageRank 算法的结果和 SSSP 几乎一致。整体上，缺失率整体较高，并且呈现周期性波动。横向对比实验数据集，Google 数据集更大，因此缓存缺失率均略高于 Wikipedia 数据集，并且在 L2 缓存明显更高。纵向对比 L2 和 L3 缓存， L2 缓存的缺失率明显比 L3 要高。

3. 3 实验的数据总结

本次实验的统计数据总结如下表所示：

数据集	数据加载用时	运行 PageRank 用时	运行 SSSP 用时	内存占用	CPU 占用率
Wiki	1.45s	7.23s	1.32s	507MB	最高 82%
Google	7.32s	154.12s	43.41s	1356MB	最高 78%

4. 总结

本次实验，我学习了 Spark 环境搭建与编程、Pregel 计算框架以及 SSSP 和 PageRank 两个算法在 Pregel 框架下的实现方式，学会使用 Windows 内置的 Perform 工具对程序运行过程的各类性能指标进行跟踪、测试和分析。整个实验过程相对顺利，包括 PCM 服务的编译安装我也没有遇到困难，而对实验测试结果的分析相对来说更具备挑战性。

最后，感谢这个学期老师与助教的指导与帮助！