# 图计算实验部分

# 实验一YiTu安装与配置实验

【实验目的】

通过本实验，学习如何在Linux环境下配置图计算执行引擎YiTu\_XGraph的安装与使用，为后续的图数据处理和分析工作做好准备。

【实验内容】

1. 安装Linux系统。
2. 安装docker(可选)
3. 安装YiTu图计算编程框架

【实验环境】

1. 操作系统：Linux（Ubuntu）
2. 软件环境：conda,python
3. 硬件要求：至少1台计算机或虚拟机，建议配置至少4GB内存和100GB的硬盘空间。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

1. 用浏览器访问https://github.com/CGCL-codes/YiTu，下载.zip源代码文件。
2. 编译安装YiTu：

通过cmake安装：

rm-rf~/temp

version=3.18

build=0

mkdir~/temp

cd~/temp

wgethttps://cmake.org/files/v$version/cmake-$version.$build-Linux-x86\_64.sh

sudomkdir/opt/cmake

sudoshcmake-$version.$build-Linux-x86\_64.sh--prefix=/opt/cmake--skip-license

sudoln-s/opt/cmake/bin/cmake/usr/local/bin/cmake

cd~

rm-rf~/temp

\*\*(仅安装图计算部分)

定位到源码/examples/YiTu\_GNN/NDP目录下：

删除原有pybind11目录，直接下载pybind11源码

gitclone<https://github.com/pybind/pybind11.git>

cmake.

（如果报错，定位到cmake报错代码处，将py修改成pybind11即可）

make

cdnondp

make

# 图计算实验部分

## 实验二DFS实验

【实验目的】

本实验旨在通过编写和执行基于YiTu\_XGraph的DFS程序，帮助学生深入理解图计算系统的工作原理，并学会使用YiTu\_XGraph进行大规模图数据分析和处理。通过此实验，学生将能够掌握图计算的基本概念、编写简单的图算法程序以及运行它们在YiTu\_XGraph系统中。

【实验内容】

1. 基于YiTu\_XGraph实现DFS算法，掌握DFS算法和函数式编程思想。
2. 使用命令行执行DFS程序。
3. 查看程序执行结果。

【实验环境】

1. 操作系统：Linux（Ubuntu）
2. 软件环境：conda,python
3. 硬件要求：至少1台计算机或虚拟机，建议配置至少4GB内存和100GB的硬盘空间用于安装Hadoop。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

1. 接下来，将以YiTu\_XGraph中的BFS算法为例，讲解如何在YiTu\_XGraph实现BFS算法并运行。

以BFS算法为例的实现细节：执行pythondemo.py--YiTu\_GNN0--methodbfs–inputGraph.bcsr--source0

后调用<root>/YiTu/examples/YiTu\_GNN/NDP/ndp/bfs\_sig\_async.cu代码，需要注意，输入只支持.bcsr和.el格式的数据集

迭代方式的BFS算法的伪代码：

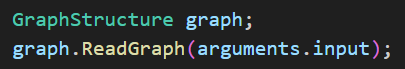
|  |  |
| --- | --- |
| **算法：**BFS() | |
| **输入：**图，指定源顶点 | |
| **输出：**图所有顶点与的深度向量 | |
|  | 初始化活跃顶点队列和下一轮的活跃顶点队列; |
|  | 初始化深度向量; |
|  | 将顶点加入; |
|  | **while**(不为空)**do** |
|  | Vertex.front(); |
|  | 出队; |
|  | **while**(&&+1)**do** |
|  | +1;//遍历到未被访问的邻居时，使邻居的深度加1 |
|  | 顶点加入队列; |
|  | **endwhile** |
|  | 交换队列和; |
|  | **endwhile** |
|  | **return**; |

1. 初始化：初始化包括分析指令、初始化图结构、初始化图状态三个阶段。

分析指令时创建ArgumentParser对象分析Linux指令的输入参数；



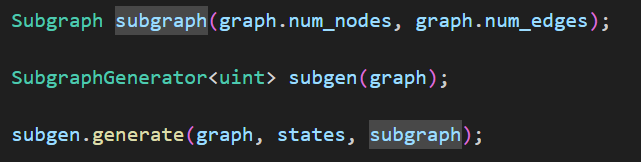
初始化图结构创建GraphStructrue对象，并根据输入的图获取图的数据，如顶点数、边数以及分布情况。并根据输入图计算各顶点的度数，以CSR格式存储。



初始化图状态根据图顶点数和算法需求创建GraphStates对象，并初始化活跃顶点

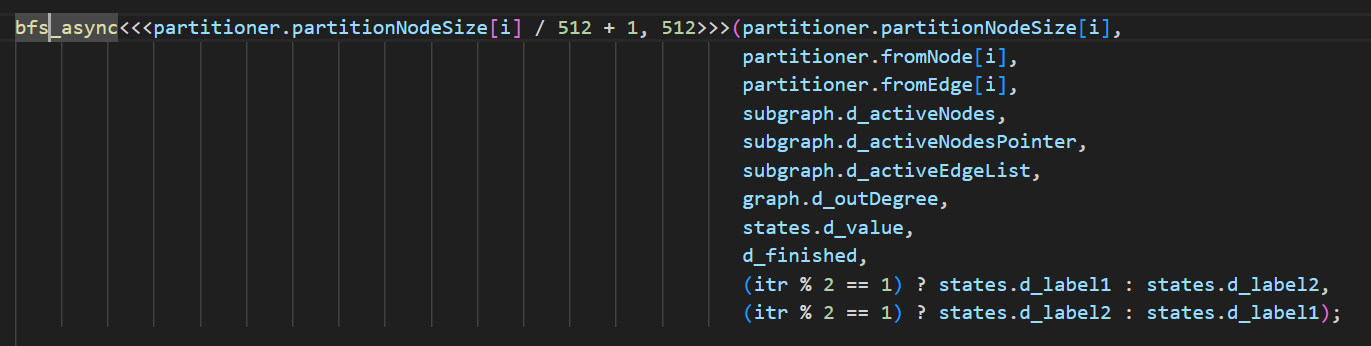


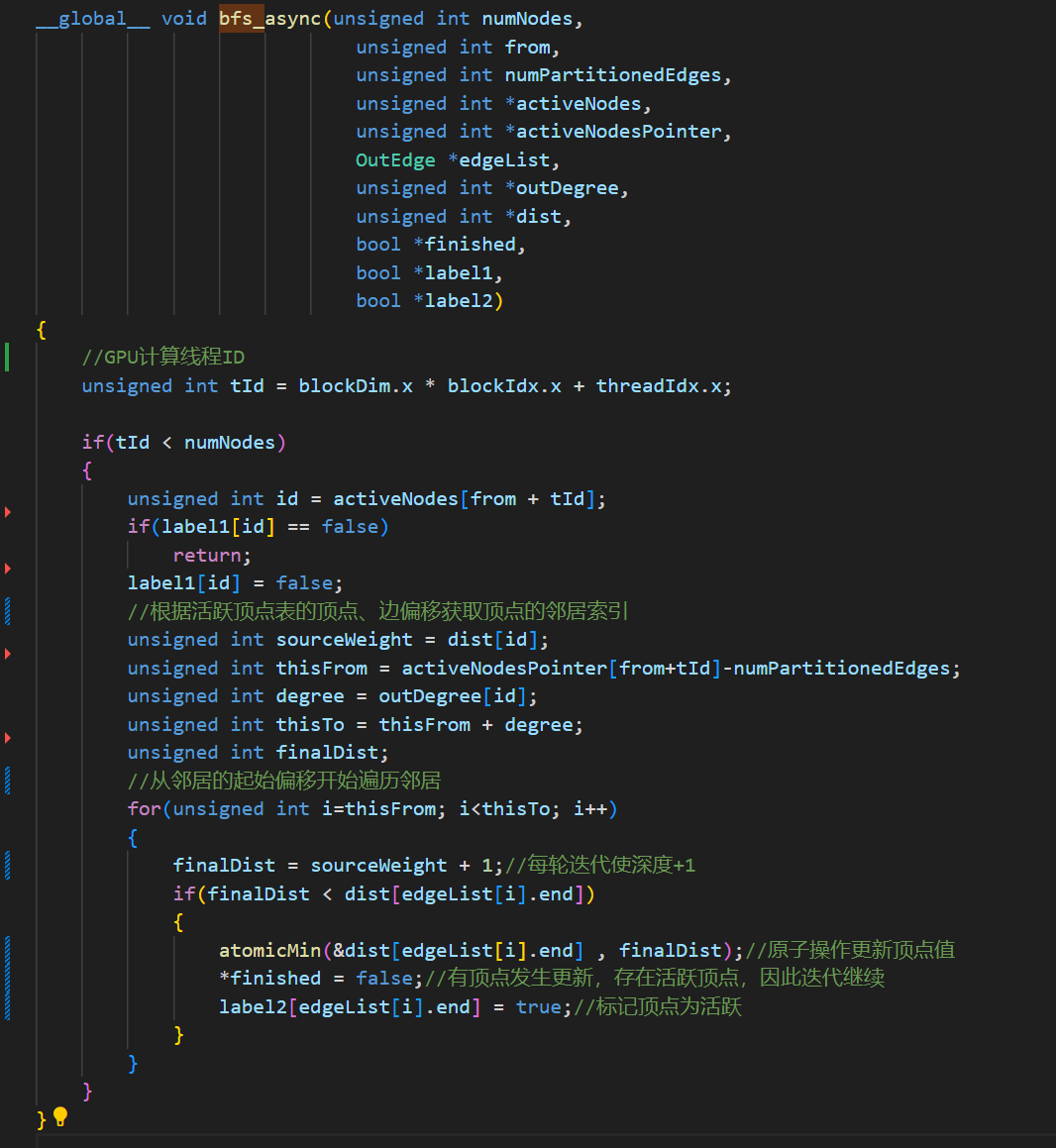
1. 子图划分：创建Sugraph、SubgraphGenerator、Partitioner等对象





1. 迭代：根据当前迭代是否存在活跃顶点进行迭代。每轮迭代执行CUDA核函数。对应核函数代码在<root>/YiTu/examples/YiTu\_GNN/NDP/shared/gpu\_kernels.cu：





运行指令和输出结果：

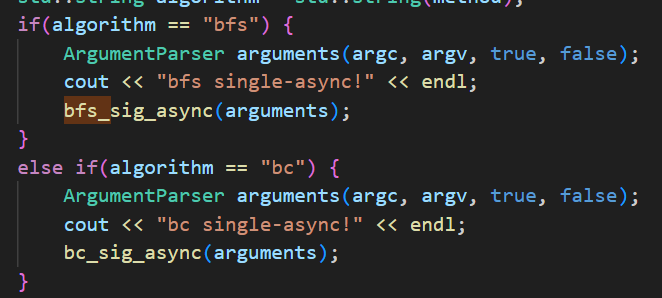
编译完成后执行Pythondemo.py–YiTu\_GNN0–method<app-name>--input<输入数据集>--source<源顶点>



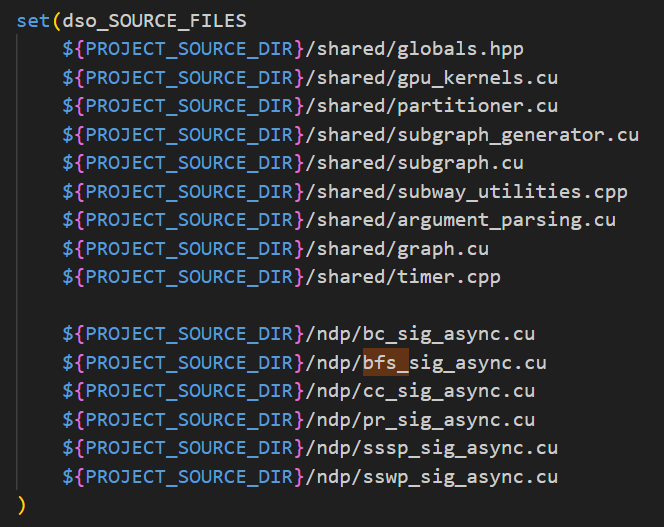
运行bfs的输出结果

接下来，请同学们将以YiTu\_XGraph中的BFS算法为例，在YiTu\_XGraph实现DFS，并成功正确地跑出结果。

（注意事项）



新添加算法后需要在<root>/YiTu/examples/YiTu\_GNN/NDP/YiTu\_GP.cpp中算法的ifelse中添加新增加的算法，使其定位到新添加的算法中。并在<root>/YiTu/examples/YiTu\_GNN/NDP/CMakeLists.txt中的set中添加新增的算法文件，并重新进行编译



# 图计算实验部分

## 实验三SCC实验

【实验目的】

本实验旨在通过编写和执行基于YiTu\_XGraph的SCC程序，帮助学生深入理解图计算系统的工作原理，并学会使用YiTu\_XGraph进行大规模图数据分析和处理。通过此实验，学生将能够掌握图计算的基本概念、编写简单的图算法程序以及运行它们在YiTu\_XGraph系统中。

【实验内容】

1. 基于YiTu\_XGraph实现SCC算法，掌握SCC算法和函数式编程思想。
2. 使用命令行执行SCC程序。
3. 查看程序执行结果。

【实验环境】

1. 操作系统：Linux（Ubuntu）
2. 软件环境：conda,python
3. 硬件要求：至少1台计算机或虚拟机，建议配置至少4GB内存和100GB的硬盘空间用于安装Hadoop。
4. 网络连接：互联网连接，用于下载所需的软件和文档。

【实验步骤】

1. 接下来，将以YiTu\_XGraph中的CC算法为例，讲解如何在YiTu\_XGraph实现CC算法并运行。

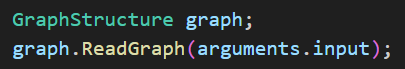
2.以CC算法为例的实现细节：执行pythondemo.py--YiTu\_GNN0--methodCC–inputGraph.bcsr--source0。与BFS算法的主要区别在于核函数的实现方式的不同

1）初始化：初始化包括分析指令、初始化图结构、初始化图状态三个阶段。

分析指令时创建ArgumentParser对象分析Linux指令的输入参数；



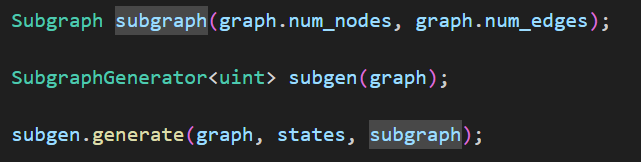
初始化图结构创建GraphStructrue对象，并根据输入的图获取图的数据，如顶点数、边数以及分布情况。并根据输入图计算各顶点的度数，以CSR格式存储。



初始化图状态根据图顶点数和算法需求创建GraphStates对象，并初始化活跃顶点

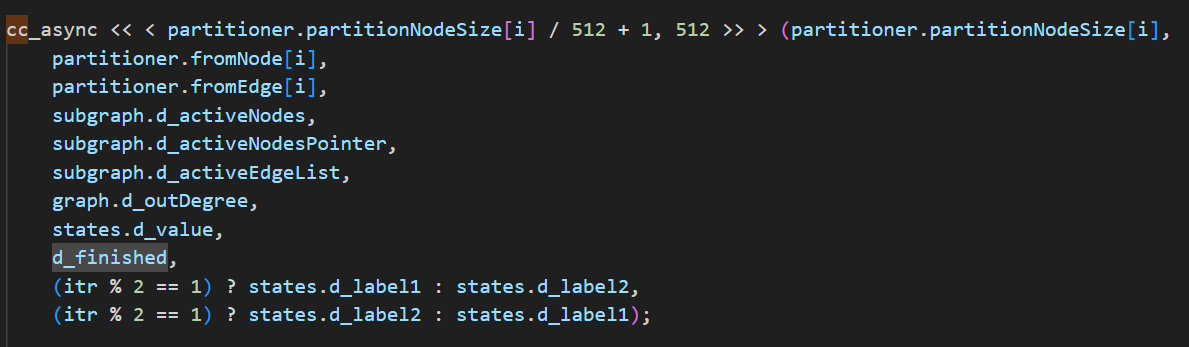


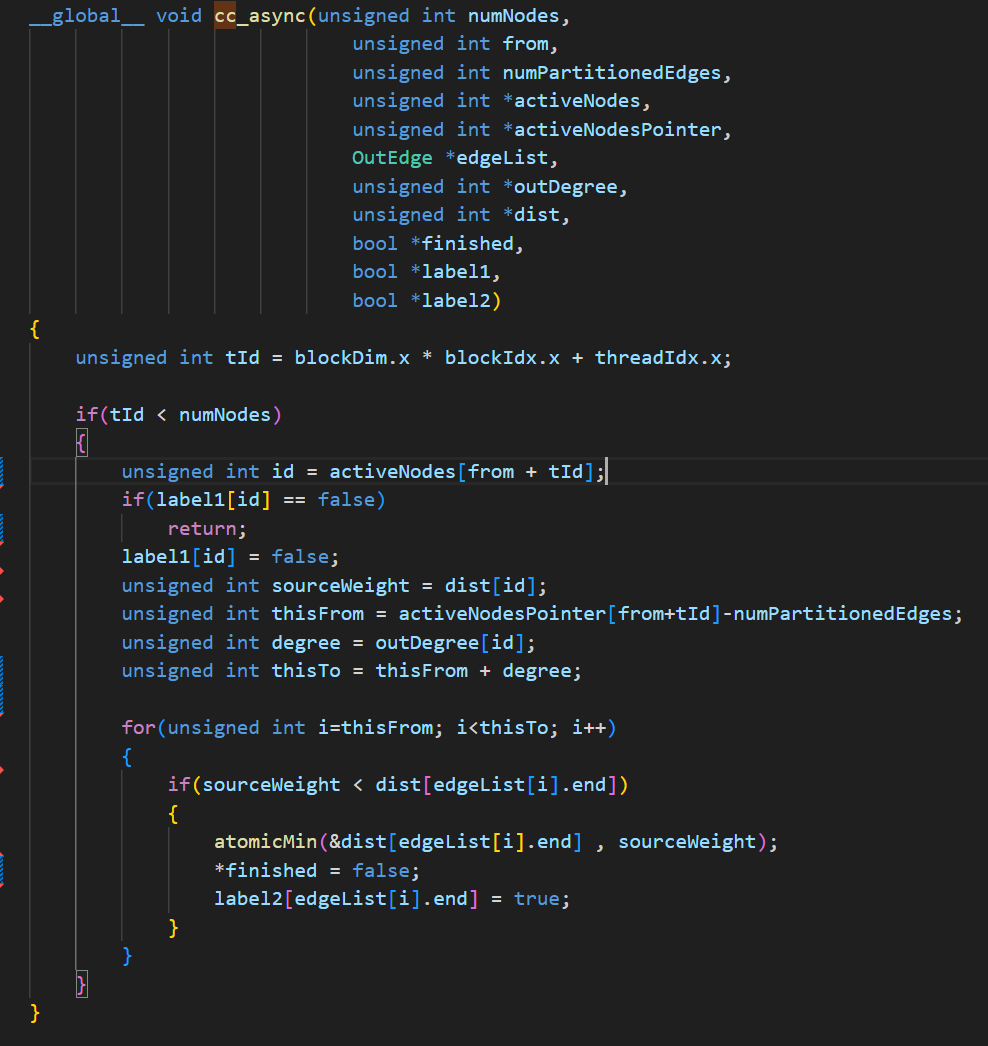
1. 子图划分：创建Sugraph、SubgraphGenerator、Partitioner等对象





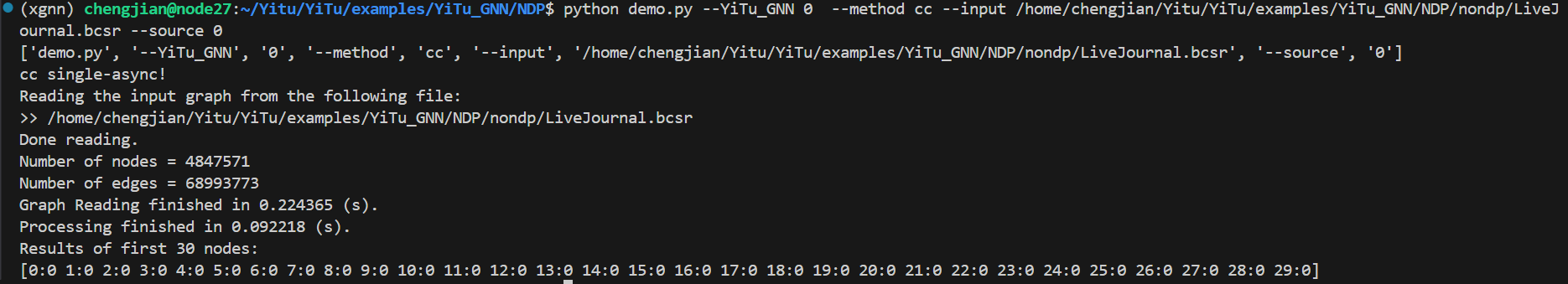
1. 迭代：根据当前迭代是否存在活跃顶点进行迭代。每轮迭代执行CUDA核函数。CC执行的核函数如下，对应代码在<root>/YiTu/examples/YiTu\_GNN/NDP/shared/gpu\_kernels.cu：





1. 运行指令和输出结果：

编译完成后执行Pythondemo.py–YiTu\_GNN0–method<app-name>--input<输入数据集>--source<源顶点>



运行CC算法的输出结果

接下来，请同学们将以YiTu\_XGraph中的CC算法为例，在YiTu\_XGraph实现SCC，并成功正确地跑出结果。

# 图计算实验部分

## 可选实验：GraphX实验

任务：由于基于YiTu的图计算实验需要安装CUDA编程库，有GPU资源的同学可以按上述的任务书来完成，没有条件的同学采用Spark的GraphX来实现图的深度优先搜索DFS和强连通分量SCC算法。

数据集：https://github.com/databricks/spark-training/tree/master/data/graphx

相关教程：<https://spark.apache.org/docs/latest/graphx-programming-guide.html>

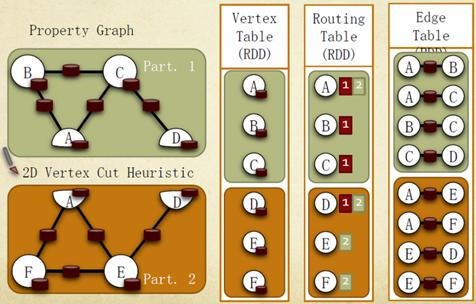
#### GraphX存储模式

Graphx借鉴PowerGraph，使用的是Vertex-Cut(点分割)方式存储图，用三个RDD存储图数据信息：

**VertexTable(id,data)**：id为Vertexid，data为Edgedata

**EdgeTable(pid,src,dst,data)**：pid为Partionid，src为原定点id，dst为目的顶点id

**RoutingTable(id,pid)**：id为Vertexid，pid为Partionid



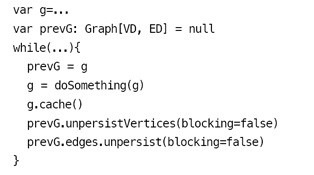
#### GraphX计算模式

GraphX的Graph类提供了丰富的图运算符，大致结构如下图所示。可以在官方[GraphXProgrammingGuide](http://spark.apache.org/docs/latest/graphx-programming-guide.html)中找到每个函数的详细说明。

对Graph视图的所有操作，最终都会转换成其关联的Table视图的RDD操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列RDD的转换过程。因此，Graph最终具备了RDD的3个关键特性：Immutable、Distributed和Fault-Tolerant，其中最关键的是Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX会有一定程度的不变顶点和边的复用优化，对用户透明。

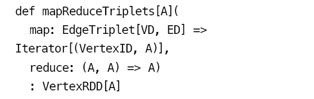
两种视图底层共用的物理数据，由RDD[Vertex-Partition]和RDD[EdgePartition]这两个RDD组成。点和边实际都不是以表Collection[tuple]的形式存储的，而是由VertexPartition/EdgePartition在内部存储一个带索引结构的分片数据块，以加速不同视图下的遍历速度。不变的索引结构在RDD转换过程中是共用的，降低了计算和存储开销。

**图的缓存机制：**每个图是由3个RDD组成，所以会占用更多的内存。相应图的cache、unpersist和checkpoint，更需要注意使用技巧。出于最大限度复用边的理念，GraphX的默认接口只提供了unpersistVertices方法。如果要释放边，调用g.edges.unpersist()方法才行，这给用户带来了一定的不便，但为GraphX的优化提供了便利和空间。参考GraphX的Pregel代码，对一个大图，目前最佳的实践是：



根据GraphX中Graph的不变性，对g做操作并赋回给g之后，g已不是原来的g了，而且会在下一轮迭代使用，所以必须cache。另外，必须先用prevG保留住对原来图的引用，并在新图产生后，快速将旧图彻底释放掉。否则，十几轮迭代后，会有内存泄漏问题，很快耗光作业缓存空间。

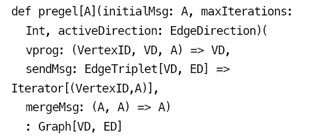
**邻边聚合机制：**mrTriplets（mapReduceTriplets）是GraphX中最核心的一个接口。Pregel也基于它而来，所以对它的优化能很大程度上影响整个GraphX的性能。mrTriplets运算符的简化定义是：

[](http://images0.cnblogs.com/blog/107289/201508/211423018165399.jpg)

它的计算过程为：map，应用于每一个Triplet上，生成一个或者多个消息，消息以Triplet关联的两个顶点中的任意一个或两个为目标顶点；reduce，应用于每一个Vertex上，将发送给每一个顶点的消息合并起来。

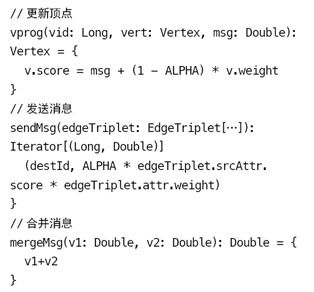
mrTriplets最后返回的是一个VertexRDD[A]，包含每一个顶点聚合之后的消息（类型为A），没有接收到消息的顶点不会包含在返回的VertexRDD中。

**Pregel模式：**GraphX中的Pregel接口，并不严格遵循Pregel模式，它是一个参考GAS改进的Pregel模式。定义如下：

[](http://cms.csdnimg.cn/article/201408/07/53e2f706971a8.jpg)

这种基于mrTrilets方法的Pregel模式，与标准Pregel的最大区别是，它的第2段参数体接收的是3个函数参数，而不接收messageList。它不会在单个顶点上进行消息遍历，而是将顶点的多个Ghost副本收到的消息聚合后，发送给Master副本，再使用vprog函数来更新点值。消息的接收和发送都被自动并行化处理，无需担心超级节点的问题。

常见的代码模板如下所示：

[](http://cms.csdnimg.cn/article/201408/07/53e2f721a71f8.jpg)

可以看到，GraphX设计这个模式的用意。它综合了Pregel和GAS两者的优点，即接口相对简单，又保证性能，可以应对点分割的图存储模式，胜任符合幂律分布的自然图的大型计算。另外，值得注意的是，官方的Pregel版本是最简单的一个版本。对于复杂的业务场景，根据这个版本扩展一个定制的Pregel是很常见的做法。

**GraphX算法工具包**：GraphX也提供了一套图算法工具包，方便用户对图进行分析。目前最新版本已支持PageRank、数三角形、最大连通图和最短路径等6种经典的图算法。这些算法的代码实现，目的和重点在于通用性。如果要获得最佳性能，可以参考其实现进行修改和扩展满足业务需求。研读这些代码，也是理解GraphX编程最佳实践的好方法。

#### 例子：基于GraphX的pagerank算法

数据集：https://github.com/databricks/spark-training/tree/master/data/graphx

测试数据为顶点数据graphx-wiki-vertices.txt和边数据graphx-wiki-edges.txt，graphx-wiki-vertices.txt顶点格式为顶点编号和网页标题，graphx-wiki-edges.txt边数据由两个顶点构成。

示例代码：

import org.apache.log4j.{Level, Logger}

import org.apache.spark.{SparkContext, SparkConf}

import org.apache.spark.graphx.\_

import org.apache.spark.rdd.RDD

object PageRank {

def main(args: Array[String]) {

//*屏蔽日志*

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

*//设置运行环境*

val conf = new SparkConf().setAppName("PageRank").setMaster("local")

val sc = new SparkContext(conf)

*//读入数据文件*

val articles: RDD[String] = sc.textFile("/home/[Hadoop](http://lib.csdn.net/base/hadoop)/IdeaProjects/data/graphx/graphx-wiki-vertices.txt")

val links: RDD[String] = sc.textFile("/home/hadoop/IdeaProjects/data/graphx/graphx-wiki-edges.txt")

*//装载顶点和边*

val vertices = articles.map { line =>

val fields = line.split('\t')

(fields(0).toLong, fields(1))

}

val edges = links.map { line =>

val fields = line.split('\t')

Edge(fields(0).toLong, fields(1).toLong, 0)

}

*//cache操作*

//val graph = Graph(vertices, edges, "").persist(StorageLevel.MEMORY\_ONLY\_SER)

val graph = Graph(vertices, edges, "").persist()

//graph.unpersistVertices(false)

*//测试*

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

*println("获取5个triplet信息")*

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

graph.triplets.take(5).foreach(println(\_))

*//pageRank算法里面的时候使用了cache()，故前面persist的时候只能使用MEMORY\_ONLY*

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

*println("PageRank计算，获取最有价值的数据")*

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

val prGraph = graph.pageRank(0.001).cache()

val titleAndPrGraph = graph.outerJoinVertices(prGraph.vertices) {

(v, title, rank) => (rank.getOrElse(0.0), title)

}

titleAndPrGraph.vertices.top(10) {

Ordering.by((entry: (VertexId, (Double, String))) => entry.\_2.\_1)

}.foreach(t => println(t.\_2.\_2 + ": " + t.\_2.\_1))

sc.stop()

}

}