BSP, Pregel and the need for Graph Processing

Patrizio Dazzi, HPC Lab
ISTI - CNR

mail: patrizio.dazzi@isti.cnr.it
web: http://hpc.isti.cnr.it/~dazzi/
Outline

• A need for Graph processing
  • existing approaches and their limits
• Google’s Pregel
  • the BSP bridging model
• Apache Spark/Bagel
  • Main features
  • Scala language
  • A couple of examples
• Conclusions
Introduction

• A need for general, distributed framework for processing big graphs
  • Web, social networks, transportation routes, ...
  • Typical problems:
    • find connected components
    • graph min-cut
    • etc...
  • Efficient processing is challenging
    • limited locality in memory accesses
    • little work for vertex
    • changing degree of parallelism
Typical Approaches

- Developing custom solutions/infrastructures
- Translating Graph algorithms to fit Map/Reduce
- Single-Machine libraries
  - e.g. BGL, LEDA, NetworkX
- Other (limited support to fault tolerance) parallel processing systems
  - BGL, CGMgraph
Pregel

- Inspired by Valiant’s Bulk Synchronous Parallel model (BSP)
- Computation as a sequence of super steps
  - vertices are first-class entities
  - communications between vertices happen only between supersteps
- Well suited by design to:
  - distributed computation
  - efficient graph processing
Pregel - History

- Proposed by Google in 2010
- Implemented in C++
- Closed source (only API are public)
- Presented in Grzegorz Malewicz et al. paper at SIGMOD
Pregel approach (1)

- **input**: a directed graph in which each vertex is uniquely identified by a vertex identifier.

- each vertex is associated with a modifiable, user defined value.

- The edges are associated with their source vertices, and each edge consists of a modifiable, user defined value and a target vertex identifier.

- **computation**: organised in a sequence of supersteps separated by global synchronisation points until the algorithm terminates.
The Pregel Approach (2)

- termination: when every vertex decide to halt
- output: a set of values explicitly output by the vertices.
  - often a directed graph isomorphic to the input
  - sometimes a set of separated values
- Approach inspired by BSP
BSP - The model in a nutshell

• Bridging Model for designing parallel algorithms
• Three stages for each superstep:
  1. concurrent computation
  2. communication
  3. barrier synchronisation
• No order between processes inside a superstep
BSP depicted
BSP - Main Features

• BSP model requires a global barrier synchronisation.
  • potentially costly, but
    • avoid deadlock or livelock: because avoid circular data dependencies.
    • simplifies fault tolerance.
  • The cost of a superstep is determined as the sum of three terms:
    • the cost of longest running computation
    • the cost of global communication
    • the cost of the barrier synchronisation
Pregel - Facts

- Strongly relies on Message Passing
  - most graph-centric algorithms do not need much more than this
- Pretty simple API
- Each vertex maintains only little information about other vertices it has to communicate to
  - less information to keep up-to-date
- Exploits Combiners to optimise the network traffic
- Exploits Aggregators for global communication
Pregel API

- Compute (msgs): the code implemented by each vertex in each superstep
- vertex _id(): to retrieve the id of the current vertex
- superstep(): the index of current superstep
- GetValue() and MutableValue(): to access and modify the value associated to the vertex
- GetOutEdgeIterator(): to retrieve information about the outlinks
- SendMessageTo (dest, message): to send messages to other vertices
- VoteToHalt(): to let a vertex specify when its computation has been terminated
Combiners

- To use when compute() can work on collapsed data instead of distinct messages
- To reduce network traffic the system can combine data belonging to different messages in a single message
- E.g. when a compute needs only the sum in input
- Need to be implemented by users
- Some algorithms by means of combiners can significantly reduce their execution time
Aggregators

- Mechanisms for global communication:
  - Each vertex can provide a value to an aggregator in superstep $S$
  - The system combines those values by reduction
  - The reduced value is made available to vertices in superstep $S+1$
- Useful for statistics, distributed queues, ...
Topology mutation

- The network topology underneath Pregel can be modified
- To limit conflicts a few rules are followed:
  - vertices removal follows edges removal
  - additions follow removal
  - edges addition follows vertices addition
  - all mutations precede calls to Compute
- Mutations are useful for certain algorithms, such as clustering
Basic Pregel Architecture

- Graph is divided into partitions
- Partition are assigned to machines
- Default assignment is made by applying the module. E.g. On k partitions the n-th vertex is assigned to the partition having n mod k as index.
- Many copies of the user program are executed by the machines
- One of these copies behaves as a master
The duties of Pregel Master

- does not process any part of the graph but
- orchestrates the computation of other machines by defining:
  - the number of partitions
  - the dispatch of input data
  - the coordination of supersteps
  - the management of the termination process
- it also coordinate the checkpointing
Open source implementations (only notable ones)

- Hadoop Hama
- Apache Giraph
  - recently ver. 1.00 has been released
- Apache Spark Bagel
  - the one we will focus on
Hadoop Hama

- Pure BSP support computing framework
- not just for graphs
- no specific support for fault tolerance, aggregators and combiners
- Built on top of Hadoop Distributed File System
- Developed in Java
Apache Giraph

• Developed by Yahoo!
• Runs on standard Hadoop
• Can be pipelined with other tasks, like Hive or MapReduce
• Synchronisation achieved by means of Apache ZooKeeper
• Fault tolerance via checkpointing
Apache Spark Bagel

- Developed by UC Berkeley
- Open source implementation of Pregel
  - Vertices and Messages as first class entities
- Run on top of Apache Spark
  - fault tolerance via Spark RDDs
- Gives its best with Scala
Apache Spark

- Initially developed at Berkeley
- Built on Hadoop Distributed FileSystem
- primitives for in-memory cluster computing
  - allows user programs to load data into a cluster's memory and query it repeatedly
  - well suited to machine learning algorithms
- Behemoth contributors, including Yahoo! and Intel
Spark - Main Features

- Deeply based on RDD
- Java, Scala, and Python APIs.
- Ability to cache (to pin) datasets in memory for interactive data analysis: extract a working set, cache it, query it repeatedly.
Need DSM?

- Cluster computing frameworks like MapReduce have been widely adopted for large-scale data analytics.
  - data parallel computations as a set of high-level operators, work distribution and fault tolerance managed automatically.
  - lack abstractions for leveraging distributed memory that maybe useful
    - applications that reuse intermediate results across multiple computations. E.g. PageRank, K-means clustering, logistic regression.
- In most current frameworks, to reuse data between computations you have to write it to an external storage system, e.g., distributed FS
  - Easily causes any application to become an I/O bound application
RDD as a solution

- fault-tolerant, parallel data structures that let users explicitly
- pin data (obtained from intermediate results) in memory
- control their partitioning to tune data placement
- and manipulate them using a set of operators.
RDD - the lineage

- RDD provides an interface based on coarse-grained data-parallel transformations
- This allows to obtain fault-tolerance
- By logging the transformations used to build a dataset
- This set of operations is called lineage
RDD vs. DSM

- RDD is written through coarse-grained transformations whereas DSM allow to access any location
- a restriction in the freedom but an enhanced performance
- in case of faults only the lost section of the dataset needs to be recovered (exploiting the information on lineage)
- RDD can exploit data locality at runtime by adapting the data partitioning
Spark Correlated Software

- Spark Streaming: A component of Spark that extends core Spark functionality to allow for real-time analysis of streaming data.
- Shark: A data warehouse system for Spark designed to be compatible with Apache Hive.
- Bagel: the Spark implementation of Google’s Pregel graph processing framework. Bagel currently supports basic graph computation, combiners, and aggregators.
Spark - Bagel

- Bagel is the Spark implementation of Pregel
- operates on a graph represented as a distributed dataset (RDD) of \((K, V)\) pairs
  - keys are vertex IDs
  - values are vertices plus their associated state.
- RDD can be either derived from
  - Scala collections
  - HDFS files
- The API is similar to Google’s Pregel
  - supports both aggregators and combiners
Scala language

- Released in 2003
- General purpose language that integrates features of
  - functional languages
  - object orientation
- Designed to
  - be scalable
  - be integrated with Java and other languages running on JVM
- Is being adopted by several big actors, like LinkedIn, Twitter, FourSquare
Brief Introduction to Scala

- Building Blocks
- Classes
- Objects
- Functional Aspects
- Traits and Mixins
Building Blocks: var

• non-final variables in Java
• if type is not declared, it is inferred from the assigned object
• can be reassigned but cannot change type

• E.g. var x = 7
Building Blocks: val

- final variables in Java
- if type is not declared, it is inferred from the assigned object
- once initialised cannot be reassigned
- must be initialised at declaration

- E.g. val x = 7
Building Blocks: def

- used to define a function
- comma-separated list of parameters in parenthesis follows the name
- the return type is specified after declaration, preceded by semicolon
- The final value of the control block is the value returned

E.g. def max (a:Int, b:Int) : Int = {
    if (x>y) x
    else y
}
Building Blocks: classes

- Has a purpose similar to Java classes
- Public by default
- Getter and Setters defined by variable declaration
- Primary constructor creates the fields
- E.g. class Coordinate(val x, val y) {
  
}
Building Blocks: auxiliary constructor

- Created as def this
- Must start with a call to a previously defined auxiliary or primary constructor
- class Coordinate(val x:Double, val y:Double) {
  def this (x:Double) = this (x, 0.0)
  def this () = this(0.0, 0.0)
}
Building Blocks: objects

- creates a singleton of a class
- no constructor parameters
- E.g. object Main {
  def main(args: Array[String]) {
    ...
  }
}
Functional Style

- Computations as the evaluation of mathematical functions
- Avoids state and mutable data
- function (def) is compiled to a functional value
- functional values can be assigned to var o val
  - if assigned to var are mutable
- can be passed as a value into another function
Traits

- A combination of Java Interfaces and Ruby Mixins
- Like objects, traits do not have constructors
- Added to a class via the extends keyword
- Additional traits can be mixed-in via the with keyword
Structure of a Bagel Program

- Definition of Vertices, Messages and Edges
- Definition of the Compute method
- Definition of the Main Object
- Optional definition of:
  - Combiners
  - Aggregators
Vertices, Edges, Messages

- @serializable class MyEdge ( val targetId: String )

  @serializable class MyVertex ( val id: String,
                                  val rank: Double,
                                  val outEdges: Seq[Edge],
                                  val active: Boolean
                                ) extends Vertex

  @serializable class MyMessage ( val targetId: String,
                                  val rankShare: Double
                                ) extends Message
Compute Method

• Compute method represents the business logic of each vertex
• the parameters are
  • the vertex itself
  • the message received during the last super step
  • the index of current superstep
• def compute ( self: MyVertex,
  msgs: Option[Seq[MyMessage]],
  superstep: Int) : (MyVertex, Iterable[MyMessage]) = {

  ...

  }

Main Object

- def main(args: Array[String]) {
  val sc = new SparkContext("local[2]", "ConnectedComponents")
  val input = sc.textFile("data.txt")

  val verts = // a function for returning vertices
  val emptyMsgs =
      sc.parallelize(List[(String, GraphMessage[Set[Int]])]())

  val algo = new HashMin
  val result = Bagel.run(sc, verts, emptyMsgs, 2)(algo.compute)

  println(result.map(v => "+%s\t%5s\n".format(v._1, v._2.rank)).collect.mkString)
}
Bagel Examples

- A few examples for showing real approaches
  - identification of connected components
    - two distinct approaches
  - Executed locally
  - Based on Scala 2.9.3 and Spark/Bagel 0.8.0
  - Code will be shown
    - inside the slides
    - inside the teacher ScalaIDE
Structure of each example

- One Object that represents the “main” of the computation
- (Custom) Classes for Edges, Vertices and Messages
- A “compute” method with a properly defined signature:

```scala
def compute(self: GraphVertex[Set[Int]],
            msgs: Option[Array[GraphMessage[Set[Int]]]],
            superstep: Int)
      : (GraphVertex[Set[Int]],
         Array[GraphMessage[Set[Int]]])
```
Connected Components

“A connected component in a graph is a subgraph in which each pair of vertices are connected one each other by path”

- Several approaches exist, both local and distributed
- Two of the distributed approaches that fit with Pregel model are:
  - Hash Min
  - Hash to All
The information owned (and shared) by each vertex

- Each vertex has a unique id
- By construction each vertex knows its own id and the ids of vertices that are directly connected to it
- Each vertex can become aware of new information by means of messages received from other vertices
- Each vertex can send information to other vertices, it is connected to, by using messages
Hash Min

- In the first iteration each vertex compute the minimum value among the ids it knows (its own id, the ids of neighbours)
- The min of the ids is then sent as a message to all its neighbours
- In the following iterations the above steps are repeated but also considering the information received inside the messages
Hash Min Implementation:
Main Object

```scala
def vertices(input: RDD[String]): RDD[(String, GraphVertex[Set[Int]])] = {
    input.map(
        line => {
            val fields = line.split('	')
            val (id, linksStr) = (fields(0), fields(1))
            val links = linksStr.split(',').map(new GraphEdge(_))
            (id, new GraphVertex[Set[Int]](id, Set(id.toInt), links, true))
        }
    ).cache
}

def main(args: Array[String]) {
    val sc = new SparkContext("local[2]", "ConnectedComponents")
    val input = sc.textFile("cc_data.txt")
    val verts = vertices(input)

    val emptyMsgs = sc.parallelize(List[(String, GraphMessage[Set[Int]])]())

    val algo = new Hash_Min
    val result = Bagel.run(sc, verts, emptyMsgs, 2)(algo.compute)

    println(result.map(v => "%s\t%s\n".format(v._1, v._2.rank)).collect.mkString)
}
```
Hash Min Implementation: 
Compute class

```scala
def compute(self: GraphVertex[Set[Int]], msgs: Option[Array[GraphMessage[Set[Int]]]], superstep: Int) : (GraphVertex[Set[Int]], Array[GraphMessage[Set[Int]]]) = {

  val halt = superstep >= 10

  def min_message(m1: GraphMessage[Set[Int]], m2: GraphMessage[Set[Int]]): GraphMessage[Set[Int]] = 
  if (((m1.value) head) < ((m2.value) head)) m1 else m2

  var minId:Set[Int] = msgs match {
    case Some(msgs) => (msgs.reduceLeft(min_message)).value
    case None => self.rank
  }
  if((minId head) > (self.rank head)) minId = self.rank

  val msgsOut = 
  if (!halt)
    self.outEdges filter { _.targetId != minId }
      map (edge => new GraphMessage(edge.targetId, minId))
  else List()

  (new GraphVertex(self.id, minId, self.outEdges, !halt), msgsOut.toArray)
}
```
Hash to All

- In the first iteration each vertex compute the union of the ids it knows (its own id, the ids of neighbours)

- The whole set is then sent to all the neighbours

- In the following iterations the above steps are repeated but also considering the information received inside the messages
Hash to All Implementation:
Main Object

```scala
def vertices(input: RDD[String]): RDD[(String, GraphVertex[Set[Int]])] = {
  input.map(
    line => {
      val fields = line.split('t
        )
      val (id, linksStr) = (fields(0), fields(1))
      val links = linksStr.split(',').map(new GraphEdge(_))

      (id, new GraphVertex[Set[Int]](id, Set.empty, links, true))
    }
  ).cache
}

def main(args: Array[String]) {
  val sc = new SparkContext("local[2]", "ConnectedComponents")
  val input = sc.textFile("cc_data.txt")
  val verts = vertices(input)

  val emptyMsgs = sc.parallelize(List[(String, GraphMessage[Set[Int]])]())

  val algo = new Hash_to_All
  val result = Bagel.run(sc, verts, emptyMsgs, 2)(algo.compute)

  println(result.map(v => "%s %s
  ".format(v._1, v._2.rank)).collect.mkString)
}
```
def compute(self: GraphVertex[Set[Int]], msgs: Option[Array[GraphMessage[Set[Int]]]], superstep: Int) : (GraphVertex[Set[Int]], Array[GraphMessage[Set[Int]]]) = {

val halt = superstep >= 10
val targets =  (self.outEdges map (edge => edge.targetId.toInt)) toSet

val neighSet:Set[Int] = msgs match {
  case Some(msgs) => ( msgs map (neighbour => neighbour.value) ) reduceLeft {(a,b) => (a | b)} union targets
  case None => targets
}

val msgsOut =
  if (!halt)
    self.outEdges map (edge => new GraphMessage(edge.targetId, neighSet))
  else
    List()

  (new GraphVertex(self.id, neighSet, self.outEdges, !halt), msgsOut.toArray)
}
Summing up

- Graph processing and analysis requires specialised solutions
- Good news: such solutions do exist
  - ...as well as tools implementing them
- Essentially based on BSP
- Spark Bagel could be a good friend for developing such solutions