REVOLUTION ANALYTICS

Parallel External Memory Algorithms Using Reference Classes

Lee Edlefsen, Ph.D. Chief Scientist
Sue Ranney, Ph.D. Chief Data Scientist

Prepared for DSC 2014, June 2014

Introduction

- In many fields, the size of data sets is increasing more rapidly than the speed of single cores, of RAM, and of hard drives
- In addition, data is being collected and stored in distributed locations
- In some applications it is also important to be able to update computations as new data is obtained
- To deal with this, there is increasing need for statistical and machine learning code that does not require all data to be in memory and that can distribute computations across cores, computers, and time
- Parallel External Memory Algorithms provide a foundation for such software

RevoPemaR

- An R package providing a framework for writing Parallel External Memory Algorithms (PEMA's)
- Uses Reference Classes
- Code can be written, tested, and run using data in memory on one computer, and then can be deployed to a variety of distributed compute contexts, using a variety of data sources
- An experimental version of this package is being released this summer as part of Revolution R Enterprise 7.2, under the Apache 2.0 license
- This framework is based on, and is very similar to, a C++ framework we have used for several years in RevoScaleR to implement extremely high performance statistical and machine learning algorithms; the main difference is that the C++ framework supports the use of multiple cores via threading

Features of code written using the RevoPemaR framework

- **Scalable**: it can process an unlimited number of rows of data in a fixed amount of RAM, even on a single core
- **Distributable:** it is distributable across processes on a single computer and across nodes of a cluster
- **Updateable**: existing results can be updated given new data
- **Portable:**
  - **With respect to platforms:** it can be executed on a wide variety of computing platforms, including the parallel and distributed platforms supported by Revolution's RevoScaleR package (Teradata, IBM Platform LSF, Microsoft HPC Server, and various flavors of Hadoop)
  - **With respect to data sources**: it can also use a wide variety of data sources, including those available in RevoScaleR

The same code will run on small and huge data, on a single core and on multiple SMP cores, on a single node and on multiple nodes.

Reference classes

- Object oriented system in R that is similar to those in Java, C++
- Fields of RC objects are mutable; they are not copied upon modification
- Methods belong to objects, can operate on the fields of the object, and all methods see the changes made by any method to a field
- Introduced in R 2.12, by John Chambers
- Is a special S4 class that wraps an environment

External memory algorithms (EMA's)

- Algorithms that do not require all data to be in RAM
- Data is processed sequentially, a chunk at a time, with intermediate results produced for each chunk
- Intermediate results for one chunk of data can be combined with those of another chunk
- Such algorithms are widely available for statistical and machine learning methods

Example external memory algorithm for the mean of a variable

- **initialize** method: sum=0, totalObs=0
- **processData** method: for each chunk of x; sum = sum + sum(x), totalObs = totalObs + length(x)
- **updateResults** method:
  - $sum_{12} = sum_1 + sum_2$
  - $totalObs_{12} = totalObs_1 + totalObs_2$
- **processResults** method: mean = sum / totalObs

Parallel external memory algorithms (PEMA's)

- PEMA's are implementations of EMA's that allow them to be executed on multiple cores and computers
- Chunks of data are processed in parallel, as well as sequentially
- To write such algorithms, the inherently sequential parts of the computation must be separated from the parallelizable parts
- For example, for iterative algorithms such a maximum likelihood (e.g. IRLS for glm), each iteration depends upon the previous one so the iterations cannot be run in parallel. However, the computations for each iteration can often be parallelized, and this is usually where most of the time is spent in any case.
- Keys to efficiency
  - avoid inter-thread and inter-process communication as much as possible
  - rapidly get chunks of data to the processData methods

Another view of a Pema

Also note that Pema is Tibetan for Lotus flower, an important Buddhist symbol: it grows in muddy water, rising and blooming above the murk to achieve enlightenment.

The RevoPemaR package provides

- PemaBaseClass, which is a Reference Class
  - Pema classes need to inherit from (contain) this
  - This class has several methods which can be overridden by child classes. The key methods that are typically overridden are:
    - initialize() – initialize fields; used on construction
    - processData() – computes intermediate results from a chunk of data
    - updateResults() – updates one pema object from another one
    - processResults() – converts intermediate results to final results
  - The compute() method provides an entry point for doing computations; the default compute() method will work for both iterative and non-iterative algorithms

The RevoPemaR package also provides

- setPemaClass() function: a wrapper around setRefClass() to create a class generator
- pemaCompute() function: a function to initiate a computation
- Example code:
  - PemaMean: variable mean
  - PemaWordCount, PemaPopularWords: text mining
  - PemaGradDescent, PemaLogitGD: a gradient descent base class and a logistic regression child

Why reference classes?

- Efficiency and lowered memory usage
  - RC's allow control over when objects are copied
  - Member variables (fields) do not have to be passed as arguments to other methods
- Encapsulation
  - The data and the methods that operate on them are bundled
  - Access methods and field locking provide control over access to fields
- Convenience
  - It is possible to do the same things in other ways in R, but RC's are more convenient
- Familiarity
  - Familiar OOP system for Java, C++, and other programmers

Typical steps in using RevoPemaR

- Write the class definition, using setPemaClass(), and inheriting from RevoBaseClass or a child class
- Specify the fields (member variables) of the class
- Override the initialize(), processData(), updateResults(), processResults() and any other required methods
- Test the individual methods as they are being written; data in memory (data frame, vector, matrix) can be used, or any of the RevoScaleR data sources
- Execute the code using the pemaCompute() function or the compute() method

The default compute() method

```
"compute" = function(inData = NULL, outData = NULL)
{
'The main computation loop; handles both single- and multi-iteration algorithms'
        for (iterLocal in seq.int(1, maxIters))
        {
                initIteration(iterLocal)
                processAllData(inData, outData)
                result <- processResults()
                if (hasConverged())
                {
                        invisible(return(createReturnObject(result)))
                }

        }
        invisible(return(createReturnObject(result)))
},
```

processAllData(inData, outData)

- processAllData() makes a pass through all of the data
- Upon return, the relevant fields of the master Pema object will contain results for all of the data
- This method is where all of the parallel/distributed computations are done
- In the initial implementation, the computations call into the RevoScaleR C++ code to loop over data and to distribute the computations, unless the data is already in memory

Overview of how things work on a single process

- The underlying framework loops over the input data, reading it a chunk (contiguous rows for selected columns) at a time
- The processData method of the Pema object is called once for each chunk of data
  - the fields of the object are updated from the data
  - in addition, a data frame may be returned from processData(), and will be written to an output data source
- processData() is called repeatedly until all of the data assigned to that process has been used
- processAllData() returns and processResults() is then called on the Pema object to convert intermediate results to final results
- hasConverged(), whose default returns TRUE, is called, and either the results are returned to the user or another iteration is started

Overview for multiple processes

- One process acts as the master, and the others as workers
- The master process controls the computations
  - This process may be different than the R process on the client computer
  - It executes the compute() method of the Pema object
- When processAllData() is called in compute()
  - The Pema object is serialized and sent to each worker, where it is deserialized and used on that worker to accumulate partial results
  - When a worker has processed all of its data, it sends its reserialized Pema object back to the master
  - The master process loops over all of the Pema objects returned by the workers and calls updateResults() to update the master Pema object

An example: computing a sample mean

To create a Pema class generator function, use the setPemaClass function. There are four basic pieces of information that need to be specified: the class name, the super classes, the fields, and the methods

```
PemaMean <- setPemaClass(
            Class = "PemaMean",
            contains = "PemaBaseClass",
            fields = list( # To be written
                        ),
            methods = list( # To be written
    ))
```

PemaMean fields

The list of field names and their types. The type “ANY” can be used to allow flexible types, but requires the author to check types

```
fields = list(
        sum = "numeric",
        totalObs = "numeric",
        totalValidObs = "numeric",
        mean = "numeric",
        varName = "character"
        ),
```

PemaMean methods: initialize

The initialize method is called when the object is constructed, and can also be called directly.

```
methods = list(
"initialize" = function(varName = "", ...)
{
    'sum, totalValidObs, and mean are all initialized to 0'
    callSuper(...) # calls the method of the parent class
    usingMethods(.pemaMethods) # for distributed computing
    # Fields are modified in a method by using the non-local assignment op
    varName <<- varName
    sum <<- 0
    totalObs <<- 0
    totalValidObs <<- 0
    mean <<- 0
},
```

PemaMean methods: processData

The processData method usually does most of the work. It takes a chunk of data and uses it to update the fields (state) of the object. It can also return a data frame of results; these will be written to an output data source.

```
"processData" = function(dataList)
{
  'Updates the sum and total observations from the current chunk of data.‘

   if (is.null(dataList[[varName]]))
            stop( "The variable ", varName, " cannot be found in the data." )

    sum <<- sum + sum(as.numeric(dataList[[varName]]), na.rm = TRUE)
    totalObs <<- totalObs + length(dataList[[varName]])
    totalValidObs <<- totalValidObs + sum(!is.na(dataList[[varName]]))
    invisible(NULL)
},
```

PemaMean methods: updateResults

The updateResults method updates the fields of one Pema object from the fields of another Pema object. This is called during distributed computations to update a master object from the results of each of the worker objects. Can also be used to update yesterday's results from results obtained from today's data.

```
"updateResults" = function(pemaMeanObj)
{
    'Updates the sum and total observations from another PemaMean object.'
    sum <<- sum + pemaMeanObj$sum
    totalObs <<- totalObs + pemaMeanObj$totalObs
    totalValidObs <<- totalValidObs + pemaMeanObj$totalValidObs
    invisible(NULL)
},
```

PemaMean methods: processResults

The processResults method converts intermediate results in a Pema object into final results.

```
"processResults" = function()
{
    'Returns the sum divided by the totalValidObs.'
            if (totalValidObs > 0)
            {
                    mean <<- sum/totalValidObs
            }
            else
            {
                    mean <<- as.numeric(NA)
            }
            return( mean )
},
```

Thank you!

- John Chambers for R Reference Classes
- R-Core Team
- R Package Developers
- R Community
- Revolution R Enterprise Customers and Testers
- Colleagues at Revolution Analytics

Contact:
lee@revolutionanalytics.com
sue@revolutionanalytics.com