SupR: Multi-threaded R Environment

Yixuan Qiu    Chuanhai Liu
Department of Statistics, Purdue University

9-th China-R Conference
Overview of Parallel Computing in R

Two Types of Parallelism: Process vs Thread

SupR: When R Meets Big Data
Overview of Parallel Computing in R

Two Types of Parallelism: Process vs Thread

SupR: When R Meets Big Data
Implemented Parallelism in R

- C/C++/FORTRAN Parallelism
- Multi-process Parallelism
- Hadoop/Spark Parallelism
- GPU Parallelism
Call parallelized C/C++/FORTRAN code from R

Pro: Simple to use (for users), excellent performance

Con: Hard to program (for developers); R objects are not thread-safe

Examples:

- OpenBLAS (Pthread/OpenMP)
- xgboost package (OpenMP)
- recosystem package (C++ 11 thread library)
- RcppParallel package (Intel TBB)
Multi-process Parallelism

- Create multiple R processes to compute simultaneously
- Use sockets or MPI to communicate between processes
- Pro: Most widely used approach in R; Can directly write R code to parallelize
- Con: Large overhead of communication; Large memory use
- Examples:
  - parallel package, `mclapply()` and `parLapply()` functions
  - Packages that use MPI, e.g. snow, Rmpi and pbdMPI
Hadoop/Spark Parallelism

- Call parallel computing frameworks such as Hadoop and Spark from R
- Pro: Based on mature framework; Capable of processing large scale data
- Con: Large communication cost with R; Large memory usage
- Examples:
  - RHIPE package (Hadoop)
  - SparkR package (Spark)
GPU Parallelism

• Use GPU to do parallel computing
• Pro: Large amount of processors with significant effect of parallelism
• Con: Demanding hardware; Few implemented algorithms
• Examples:
  • gputools package (CUDA)
  • gpuR package (OpenCL)
Outline

Overview of Parallel Computing in R

Two Types of Parallelism: Process vs Thread

SupR: When R Meets Big Data
Two Types of Parallelism

Multi-process

Process 1
- Code and Data
- Thread 1

Process 2
- Code and Data
- Thread 1

Multi-thread

Process
- Shared Code and Data
- Thread 1
- Thread 2
- ...
Process vs Thread

- A **process** may contain multiple **threads**
- **Threads** are more lightweight than **processes**
- **Threads** in the same **process** can share resources
- Communication between **threads** has less overhead
Parallelism in R

• Most of the implementations in R are based on multi-process parallelism
• Multi-threading generally relies on C/C++/FORTRAN/Java
• Multi-threading has many advantages over multi-processing
• However, it is quite challenging to enable multi-threading in R
  • Interpreter, memory allocation and garbage collection are not thread-safe
  • Multi-threading code may break the internal structure and operation mechanism of R
  • Modification of R source code must be made to enable multi-threading
Overview of Parallel Computing in R

Two Types of Parallelism: Process vs Thread

SupR: When R Meets Big Data
• Support for big data analysis is not fully ideal in R yet
• One of the issues is its limited parallelism mechanism
• An ideal platform for big data analysis:
  • Multi-threading on single machine
  • Distributed computing on clusters
  • Good interactivity
  • Comprehensive community support
Born of SupR

- SupR is a modified R that supports multi-threading and distributed computing
- Developed by Prof. Chuanhai Liu in Department of Statistics, Purdue University
- Targeting on building a platform for big data analysis
- Key features:
  - Keep R syntax and internal structure unchanged
  - Support for multi-threading
  - Support for Spark-like distributed computing on clusters
  - Support for distributed file system
Multi-threading in SupR

- SupR makes modification to source code of R
- Mutex locks are added to areas of code that may cause thread conflicts
- Complete operations on thread creation, query, interruption, synchronization and cancellation
Key Functions

• `new.thread()`: Create threads
• `start.thread()`: Start threads
• `sync.eval()`: Synchronized evaluation of code
• `wait()`: Make threads sleep and wait for signal
• `notify()`: Wake up threads
Example - Simple Parallelism

• Put expressions that need to be parallelized into two threads

```r
set.seed(123)
n = 10
A = matrix(rnorm(n^2), n)
B = matrix(rnorm(n^2), n)
th1 = new.thread({
    C1 <<- A %*% B[, 1:(n/2)]
})
th2 = new.thread({
    C2 <<- A %*% B[, (n/2+1):n]
})
start.thread(th1); start.thread(th2)
ls()
## [1] "A"  "B"  "C1"  "C2"  "n"  "th1"  "th2"
```
Example - Synchronized Evaluation

- For all `sync.eval()` commands that have the same `x` object, at any time only one of them can be executed

```r
x = "any object"
threads = vector("list", 5)
for(i in 1:5) {
    threads[[i]] = new.thread(sync.eval(x, {
        cat(current.thread(), ": ", as.character(Sys.time()), 
        "\n", sep = "")
        thread.sleep(2)
    })))
}
for(i in 1:5) start.thread(threads[[i]])
## thread_1: 2016-05-25 09:21:24
## thread_3: 2016-05-25 09:21:27
## thread_5: 2016-05-25 09:21:29
## thread_4: 2016-05-25 09:21:31
## thread_2: 2016-05-25 09:21:33
```
Example - Thread Communication (1)

- `wait()` makes a thread to sleep until it is awakened by the `notify()` function
- **Master thread**
  1. Create master thread and print start-up message
  2. Notify workers and wait for the finish of job
  3. Print ending message

```r
sync.m = "master barrier"
sync.w = "worker barrier"
threads = vector("list", 3)
count = 0
master = new.thread(sync.eval(sync.m, {
  cat("Thread M: Program starts\n"
  ## Notify workers starting to work
  sync.eval(sync.w, notify(sync.w, all = TRUE))
  ## Wait for workers to finish
  wait(sync.m)
  cat("Thread M: Program ends\n")
}))
```
Worker threads

1. Create worker threads and wait for master’s notification
2. Increment count by 1 and print message; Only one thread can work at any time
3. When job is finished (count increases to 10), notify master thread and end

```r
work = function() {
    sync.eval(sync.w, wait(sync.w))
    while(TRUE) {
        sync.eval(sync.w, {
            if(count < 10) {
                count <<- count + 1
                cat(current.thread(), "is working, count =", count, "\n")
            } else break
        })
    }
    sync.eval(sync.m, notify(sync.m))
}
for(i in 1:3) new.thread(work(), start = TRUE)
```
• Output

```
start.thread(master)
## Thread M: Program starts
## thread_3 is working, count = 1
## thread_4 is working, count = 2
## thread_2 is working, count = 3
## thread_3 is working, count = 4
## thread_4 is working, count = 5
## thread_2 is working, count = 6
## thread_3 is working, count = 7
## thread_4 is working, count = 8
## thread_2 is working, count = 9
## thread_3 is working, count = 10
## Thread M: Program ends
```
Development Status

• Currently in experimental development
• Final version will be released with open source
• Binary version can be downloaded from project webpage
  • http://www.stat.purdue.edu/~chuanhai/SupR/internal/
• Username: monkey, Password: 2016
• Project page includes relevant documentation
Questions?
Thanks!