# Rthreads -- Threads for R!

# Filling an Important Gap

- *Threading* is a major mechanism for parallel operations in the world of computing.
- Using C/C++, a popular threading library is OpenMP.
- R does not have native threading.
- Lack of threading in R means that developers of fast packages like **data.table** must rely on threads at the C++ level (and only implementing specific functions).
- Alternative to threads is *message-passing*, e.g. R's **parallel** package, including via **foreach** interface.
- Threaded coding tends to be clearer and faster, compared to message-passing.
- Thus having a threads capability in R would greatly enhance R's capabilities in parallel processing.

# What Is the Difference between Threading and Message Passing?

- Say we are on a quad-core machine. Here is an overview of the two paradigms (lots of variants):
- Message passing:
  - We would write 2 functions, a worker and a manager. There would be 4 copies of the worker code, running fully independently.
  - The manager would communicate with the workers by sending/receiving messages across the network.
  - The manager would, say, break a task into 4 chunks, and send the chunks to the workers.
  - Each worker would work on its chunk, then send the result back to the manager, which would combine the received results.
  - Example: Sorting. Each worker sorts its chunk, then the manager merges the results to obtain the final sorted vector.
- Threading:
  - There would be no manager, just 4 copies of worker code (different code from the message-passing case).
  - The threads assign themselves chunks of work to do, instead of being assigned by manager code.
  - Similarly, the threads combine the results of work on chunks on their own, rather than manager code doing this.
  - The workers operate mostly independently, but interact via variables in shared RAM. One thread might modify a shared variable **x**, and then another thread might read the new value.
  - Say **x** is shared and **y** is nonshared. Then there is only one copy of **x** but 4 copies of **y**.
  - By the way, what about Python? It does have threading capability, but has always been useless for parallel computation, as its *Global Interpreter Lock* disallows more than one thread running at a time. However, the newest version of Python has an experimental GIL-less option.

# Rthreads Implementation

- True physical shared RAM, via **bigmemory** package.
- 100% R; no C/C++ code to compile.
- Parallel computation under the shared-memory paradigm.
- Coding style and performance essentially equivalent to that of "threaded R" if it were to exist.
- Each thread is a separate instance of R.
- Formerly the **Rdsm** package, but fully rewritten.

## How Rthreads Works

- The sole data type is matrix, a **bigmemory** constraint. A matrix must be explicitly written with two (possibly empty) subscripts, e.g. **x[3,2]**, **x[,1:5]**, **x[,]**.
- The related **synchronicity** package provides mutex support.
- Each thread must run in its own terminal window.
    - This is to facilitate debugging application code.

      Note: Parallel programming is hard, in any form, and thus one may spend much more time debugging code than writing it.
    - Use **tmux** if screen space is an issue. See below.
- Run **rthreadsSetup** in the first window, then run **rthreadsJoin** in each window.

  Now call your application function code in each window.
- Some in the computing field believe that one should avoid having global variables. However, globals are the essence of threading. In R, the generally accepted implementation of globals is to place them in an R environment, which we do here:
    - Shared variables are in the environment **sharedGlobals**.
    - No-shared variables are in the environment **myGlobals**.

## Installation

```
devtools::install_github('https://github.com/matloff/Rthreads')
```

## Getting Acquainted with Rthreads (5 minutes)

Create two terminal windows, and in each of them start R and load **Rthreads**. Then go through the sequence below, where a blank entry means to not type anything in that window in that step.

| Step | Window 0 | Window 1 |
|---|---|---|
| 1 | rthreadsSetup(2) | |
| 2 | rthreadsJoin() | rthreadsJoin() |
| 3 | rthreadsMakeSharedVar('x',1,1,initVal=3) | |
| 4 | | rthreadsAttachSharedVar('x') |
| 5 | sharedGlobals[['x']]1,1] | sharedGlobals[['x']][1,1] |

| Step | Window 0 | Window 1 |
|---|---|---|
| 6 | | sharedGlobals[['x']][1,1] <- 8 |
| 7 | sharedGlobals[['x']]1,1] | sharedGlobals[['x']][1,1] |
| 8 | sharedGlobals$x[1,1] <- 12 | |
| 9 | sharedGlobals[['x']]1,1] | sharedGlobals[['x']][1,1] |
| 10 | myGlobals[['myID']] | myGlobals[['myID']] |

: Get-acquainted run-through

(R environments use R list notation. **myGlobals[['myID']** and **myGlobals$myID** are equivalent.)

Here is what happens:

- Step 1: Set up 2 threads (which will be numbered 0 and 1).
- Step 2: Each thread checks in. The first thread has already done so, so just one thread checks in this case. Note that each thread, upon checking in, will then wait for the others to check in.
- Step 3: Thread 0 creates a shared variable **x**, initial value 3.
- Step 4: Thread 1 attaches **x**.
- Step 5: We view **x** from each thread, seeing the value 3.
- Step 6: Thread 1 changes the value of **x** to 8.
- Step 7: We confirm that both threads now see the value 8 for **x**.
- Step 8: Thread 0 changes the value of **x** to 12.
- Step 9: We confirm that both threads now see the value 12 for **x**.
- Step 10: We confirm that the thread IDs are 0 and 1. Note that they are not shared.

# Example 1: Sorting many long vectors

(Note: The code for all of the examples here are in *_inst/examples_.)

We have a number of vectors, each to be sorted.

```
# threads configuration: run
# rthreadsSetup(nThreads=2,
#    sharedVars=list(nextRowNum=c(1,1,3),m=c(10,100000000)))

setup <- function()  # run in "manager thread"
{
   # generate vectors to be sorted, of different sizes
   tmp <- c(30000000,70000000)
   set.seed(9999)
   nvals <- sample(tmp,10,replace=TRUE)  # 10 vectors to sort
   for (i in 1:10) {
      n <- nvals[i]
      sharedGlobals$m[i,1:(n+1)] <- c(n,runif(n))
   }
}

doSorts <- function()  # run in all threads, maybe with system.time()
{

   rowNum <- myGlobals$myID+1  # my first vector to sort
```

```
      while (rowNum <= nrow(sharedGlobals$m)) {
         # as illustration of parallel operation, see which threads execute
         # sorts on which rows
         print(rowNum)
         n <- sharedGlobals$m[rowNum,1]
         x <- sharedGlobals$m[rowNum,2:(n+1)]
         sharedGlobals$m[rowNum,2:(n+1)] <- sort(x)
         rowNum <- rthreadsAtomicInc('nextRowNum')
      }

      rthreadsBarrier()

   }
```

To run, say with just 2 threads:

1. Let's refer to the 2 terminal windows as W0 and W1.

2. Start **Rthreads**:

   In W0, run

   ```
   rthreadsSetup(nThreads=2,
      sharedVars=list(nextRowNum=c(1,1,3),m=c(10,100000000)))
   ```

   This sets up 2 threads (running in the 2 windows), and 2 shared variables: **nextRowNum**, a scalar, and **m**, the latter being our matrix of rows to be sorted, 10 rows of length 100000000 each.

   In both windows, run

   ```
   rthreadsJoin()
   ```

   Here each thread "checks in," attaches the shared variables, sets its ID, and then waits until all the threads have joined.

3. Set up and run app:

   In W0, run **setup()** to generate the data.

   In both W0 and W1, run **doSorts()** to do the sorting.

   Sorted rows are now available in **sharedGlobals$m**.

Overview of the code:

- Each thread works on one row of **m** at a time.
- When a thread finishes sorting a row, it determines the next row to sort by inspecting the shared variable **nextRowNum**. It increments that variable by 1, using the old value as the row it will now sort.
- The incrementing must be done *atomically*. Remember, **nextRowNum** is a shared variable. Say its value is currently 7, and two threads execute the incrementation at about the same time. We'd like one thread to next sort row 7 and the other to sort row 8, with the new value of **nextRowNum** now being 9. But if there is no constraint on

simultaneous access, both threads may get the value 7, with **nextRowNum** now being 8 (called a *race condition*). Use of **rthreadsAtomicInc** ensures that only one thread can access **nextRowNum** at a time.

- Here is the internal code for **rthreadsAtomicInc**:

```
rthreadsAtomicInc <- function(sharedV,mtx='mutex0',increm=1)
{
   mtx <- get(mtx,envir=sharedGlobals)
   synchronicity::lock(mtx)
   shrdv <- get(sharedV,env=sharedGlobals)
   oldVal <- shrdv[1,]
   newVal <- oldVal + increm
   shrdv[1,1] <- newVal
   synchronicity::unlock(mtx)
   return(oldVal)
}
```

  The key here is use of a *mutex* (short for "mutual exclusion"), which can be locked and unlocked. While locked, no other thread is allowed to enter the given section of code, i.e. the lines beteween the lock and unlock operations. (Termed a *critical section* in the parallel processing field.)

  If one thread has locked the mutex and another thread reaches the lock line, it will be blocked until the mutex is unlocked.

- As noted earlier, in threads programming, the threads assign work to themselves, instead of manager code doing so. Here this is done via the shared variable **nextRowNum**.

  Another possibility would be to have threads *pre*-assign work to themselves. With 10 rows and 2 threads, for instance, the first thread could work on rows 1 through 5, with the second handling rows 6 to 10. But this may not work well in settings with *load imbalance*, where some rows require more work than others (as with our test case here).

## Example 2: Shortest paths in a graph

We have a *graph* or *network*, consisting of people, cities or whatever, with links between some of them, and wish to find the shortest path from all vertices A to a vertex B. This ia a famous problem, with lots of applications. Here we take a matrix approach, with a parallel solution via **Rthreads**.

We treat the case of *directed* graphs, meaning that a link from vertex i to vertex j does not imply that a link exists in the opposite direction. For a graph of v vertices, the *adjacency matrix* M of the graph is of size v X v, with the row i, column j element being 1 or 0, depending on whether there is a link from i to j. We also assume the matrix is a *directed acylic graph* (DAG), meaning that any path leading out of vertex i cannot return to i.

Here is the code:

```
# threads configuration: run
#    rthreadsSetup(nThreads=2)

# algorithm assumes a Directed Acyclic Graph (DAG); for test cases, an
# easy

setup <- function(preDAG,destVertex)  # run in "manager thread"
{
   library(bnlearn)
   # to generate a DAG, take any data frame and run it through, say,
   # bnlearn:hc
```

```
    adj <- amat(hc(preDAG))
    n <- nrow(adj)
    rthreadsMakeSharedVar('adjm',n,n,initVal=adj)
    rthreadsMakeSharedVar('adjmPow',n,n,initVal=adj)
    # if in row i = (u,v), u is not 0 then it means this path search ended
    # after iteration u; v = 1 means reached the destination, v = 2
    # means no paths to destination exist
    rthreadsMakeSharedVar('done',n,2,initVal=rep(0,2*n))
    rthreadsMakeSharedVar('imDone',1,1,initVal=0)
    rthreadsMakeSharedVar('NDone',1,1,initVal=0)
    rthreadsMakeSharedVar('dstVrtx',1,1,initVal=destVertex)
    rthreadsInitBarrier()
    return()
 }

 findMinDists <- function()
    # run in all threads, maybe with system.time()
 {
    if (myGlobals$myID > 0) {
       rthreadsAttachSharedVar('adjm')
       rthreadsAttachSharedVar('adjmPow')
       rthreadsAttachSharedVar('done')
       rthreadsAttachSharedVar('NDone')
       rthreadsAttachSharedVar('dstVrtx')
    }
    destVertex <- sharedGlobals$dstVrtx[1,1]
    n <- nrow(sharedGlobals$adjm[,])
    myRows <-
       parallel::splitIndices(n,myGlobals$info$nThreads)[[myGlobals$myID+1]]
    mySubmatrix <- sharedGlobals$adjm[myRows,]

    # find "dead ends," vertices to lead nowhere
    tmp <- rowSums(sharedGlobals$adjm[,])
    deadEnds <- which(tmp == 0)
    sharedGlobals$done[deadEnds,1] <- 1
    sharedGlobals$done[deadEnds,2] <- 2
    # and don't need a path from destVertex to itself
    sharedGlobals$done[destVertex,] <- c(1,2)
    deadEndsPlusDV <- c(deadEnds,destVertex)

    imDone <- FALSE
    for (iter in 1:(n-1)) {
       rthreadsBarrier()
       if (sharedGlobals$NDone[1,1] == myGlobals$info$nThreads) return()
       if (iter > 1 && (iter <= n-1))
          sharedGlobals$adjmPow[myRows,] <-
             sharedGlobals$adjmPow[myRows,] %*% sharedGlobals$adjm[,]
       if (!imDone) {
          for (myRow in setdiff(myRows,deadEndsPlusDV)) {
             if (sharedGlobals$done[myRow,1] == 0) {
                # this vertex myRow not decided yet
                if (sharedGlobals$adjmPow[myRow,destVertex] > 0) {
                   sharedGlobals$done[myRow,1] <- iter
                   sharedGlobals$done[myRow,2] <- 1
                } else {
                   currDests <- which(sharedGlobals$adjmPow[myRow,] > 0)
                   # check subset
                   currDestsEmpty <- (length(currDests) == 0)
                   if (currDestsEmpty ||
                       !currDestsEmpty &&
                          identical(intersect(currDests,deadEnds),currDests))  {
```

```
                     sharedGlobals$done[myRow,1] <- iter
                     sharedGlobals$done[myRow,2] <- 2
                  }
               }
            }
         }
         if (sum(sharedGlobals$done[myRows,1] == 0) == 0) {
            imDone <- TRUE
            rthreadsAtomicInc('NDone')
         }
      }
   }

}
```

As a test run, do

```
data(svcensus)
setup(svcensus,4)
```

in the first window. This generates a DAG for our test, and specifies that we are interested in vertex 4 as the destination, i.e. "B.". Then run

```
findMinDists()
```

in all windows. Inspect **sharedGlobals$done[,]** to check the results; see description of that matrix below.

A key property is that the k-th power of M tells us whether there is a k-link path from i to j, according to whether the row i, column j element in the power is nonzero. The matrix powers are computed in parallel, with each thread being responsible for a subset of rows:

```
n <- nrow(sharedGlobals$adjm[,])
myRows <-
   parallel::splitIndices(n,myGlobals$info$nThreads)[[myGlobals$myID+1]]
mySubmatrix <- sharedGlobals$adjm[myRows,]
```

If say the matrix has 100 rows and we have 2 threads, the first thread will assign itself rows 1 to 50, and the second will handle rows 51 to 100.

Here we have used the fact that in a matrix product W = UV, row i of W is equal to the product of row i of U with V.

As noted, we will find the shortest distance from all vertices A to a given destination B, which is **destVertex** in the code. We will store results in the shared matrix **done**, and in fact that object will contain the final results in the end. If row i in **done** has value (r,s), it means that in iteration r our search for paths from i to the destination ended. If s = 1, that means the destination was reached in a path of r links; if s = 2, it is impossible to get from i to the destination.

Since we assume the graph is *acyclic*, any path can be of length at most **n-1** links, and typically will be shorter. Not only might a path reach the destination with fewer links, but also a path might end at an *absorbing vertex*, one with no outgoing links. We refer to them as "dead ends" in the code.

Taking all this into account, we see that even though our **for** loop has a nominal number of iterations **n-1**, we often will exit the loop well before that.

Here is where the main work is done:

```
for (myRow in setdiff(myRows,deadEndsPlusDV)) {
   if (sharedGlobals$done[myRow,1] == 0) {
      # this vertex myRow not decided yet
      if (sharedGlobals$adjmPow[myRow,destVertex] > 0) {
         sharedGlobals$done[myRow,1] <- iter
         sharedGlobals$done[myRow,2] <- 1
      } else {
         currDests <- which(sharedGlobals$adjmPow[myRow,] > 0)
         # check subset
         currDestsEmpty <- (length(currDests) == 0)
         if (currDestsEmpty ||
             !currDestsEmpty &&
                identical(intersect(currDests,deadEnds),currDests))  {
            sharedGlobals$done[myRow,1] <- iter
            sharedGlobals$done[myRow,2] <- 2
         }
      }
   }
}
```

Here we exclude the dead ends, and also the destination vertex.

The "if" section here is straightforward; if we find that on this iteration there is at least some path, of those traversed so far, to the destination, we update **done** accordingly.

The "else" part looks at the nonzero elements of the power matrix in the current iteration. These tell us all the vertices at which we could be after hopping this number of links. The question is then, are all such vertices dead ends? If so, we've found that it is impossible to reach the destination from vertex **myRow**, and we update **done** accordingly.

# Example 3: Missing value imputation

Here the application is imputation of NAs in a large dataset. Note that I've kept it simple, so as to best illustrate the parallelization principles involved; it is neither an optimal way to do imputation nor an optimally speedy parallelization of the given imputation problem.

We have a large data frame with numerical entries, but with NA values here and there. A common imputation approach for a given column is to run some kind of regression method (parametric or nonparametric) to predict this column from the others, then replace the NAs by the predicted values.

We will do this column by column, with each thread temporarily saving its imputed values temporarily rather than immediately writing them back to the dataset. Only after all threads have computed imputations in the given round do we update the actual dataset.

Since we are working column by column, this means earlier imputations can be employed in the regression actions of later columns, hopefully improving imputation accuracy. (Again, this may or may not be the case, but that is the motivation behind this imputation algorithm.) See comments in the code for further discussion.

The purpose of this example is mainly to illustrate the concept of a *barrier*, an important threads concept. When a thread reaches that line, it may not proceed further until *all* threads have reached the line. Why is this needed here? Actually, we don't need it in this particular case, but I've included it to explain the barrier concept, as follows.

Here is the code:

```
# threads configuration: run
#    rthreadsSetup(nThreads=2)

# NA imputation, simple use of linear regression, each column's NAs
# replaced by fitted values

# note that NA elements imputed in one column will be used as inputs to
# imputation in later columns (after the curren "round"; see below)

# mainly for illustrating barriers; could be made faster in various ways

# for more of a computational-time challenge, try say k-NN instead of
# linear regresion

setup <- function()  # run in "manager thread"
{
   data(NHISlarge)
   nhis.large <- regtools::factorsToDummies(nhis.large,dfOut=FALSE)
   nhis.large <- nhis.large[,-(1:4)]  # omit ID etc.
   z <- dim(nhis.large)
   nr <- z[1]
   nc <- z[2]
   rthreadsMakeSharedVar('dta',nr,nc,initVal=nhis.large)
   rthreadsInitBarrier()
}

doImputation <- function()
{
   if (myGlobals$myID > 0) {
      rthreadsAttachSharedVar('dta')
   }
   nc <- ncol(sharedGlobals$dta)
   nThreads <- myGlobals$info$nThreads

   # in each round, each thread works on one column; they then update
   # the data
   nRounds <- ceiling(nc/myGlobals$info$nThreads)
   numPerRound <- floor(nc/nRounds)
   for (i in 1:nRounds) {

      myColNum <- (i-1)*numPerRound + myGlobals$myID + 1

      # impute this column, if needed
      myImputes <- NULL
      if (myColNum <= nc) {
         print(myColNum)
         NAelements <- which(is.na(sharedGlobals$dta[,myColNum]))
         if (length(NAelements) > 0) {
            lmOut <-
               lm(sharedGlobals$dta[,myColNum] ~ sharedGlobals$dta[,-myColNum])
            # note: if a row has more than 1 NA, imputed value
            # will still be NA
            imputes <- lmOut$fitted.values[NAelements]
            myImputes <-
               list(colNum=myColNum,imputes=imputes,NAelements=NAelements)
         }
      }

      # update data, where needed
```

```
         rthreadsBarrier()  # can't change dta while possbily still in use

         if (!is.null(myImputes)) {
            colNum <- myImputes$colNum
            NAelements <- myImputes$NAelements
            imputes <- myImputes$imputes
            sharedGlobals$dta[NAelements,colNum] <- imputes
         }

         rthreadsBarrier()  # some threads may still be writing to dta

      }

   }
```

To run the code, first run **setup** in one window, then **doImputation** in all windows. The shared object **sharedGlobals$dta** contains the dataset throughout the code, and upon completion the imputed dataset will there.

Work is done on groups of columns, called "rounds" here.

```
   nc <- ncol(sharedGlobals$dta)
   nThreads <- myGlobals$info$nThreads

   # in each round, each thread works on one column; they then update
   # the data
   nRounds <- ceiling(nc/myGlobals$info$nThreads)
   numPerRound <- floor(nc/nRounds)
   for (i in 1:nRounds) {
```

Within a round, each thread checks its assigned column for NAs, and performs the imputation (if any) in **myImputes**:

```
   myImputes <- NULL
   if (myColNum <= nc) {
      print(myColNum)
      NAelements <- which(is.na(sharedGlobals$dta[,myColNum]))
      if (length(NAelements) > 0) {
         lmOut <-
            lm(sharedGlobals$dta[,myColNum] ~ sharedGlobals$dta[,-myColNum])
         # note: if a row has more than 1 NA, imputed value
         # will still be NA
         imputes <- lmOut$fitted.values[NAelements]
         myImputes <-
            list(colNum=myColNum,imputes=imputes,NAelements=NAelements)
      }
   }
```

At the end of a round, each thread will update the dataset with the imputations it found. However, it must first make sure all threads are done with their imputation processes; otherwise, this thread might write to the dataset while another thread is still using the old version of the dataset. This is accomplished by the barrier operation:

```
         myImputes <-
            list(colNum=myColNum,imputes=imputes,NAelements=NAelements)
```

```
      }
   }

   # update data, where needed
   rthreadsBarrier()  # can't change dta while possbily still in use
```

Similarly, after performing the update and going on to the next round, we must first make sure all threads are done with their update operations, thus another barrier:

```
   if (!is.null(myImputes)) {
      colNum <- myImputes$colNum
      NAelements <- myImputes$NAelements
      imputes <- myImputes$imputes
      sharedGlobals$dta[NAelements,colNum] <- imputes
   }

   rthreadsBarrier()
```

## Regarding Repeat Runs

Objects created by **bigmemory** are persistent until removed or until the R session ends. This can have implications if you do a number of runs of an **Rthreads** application. Be sure your application's **setup** function re-initializes **bigmemory** objects as needed.

Note too that **myID** and **info** are also persistent.

## Facilitating Rthreads Use via 'screen' or 'tmux'

In using **Rthreads**, one needs a separate terminal window for each thread. Some users may have concerns over the screen real estate that is used. The popular Unix (Mac or Linux) **screen** and **tmux** utilities can be very helpful in this regard. Basically, they allow multiple terminal windows to share the same screen space.

Methods for automating the process of setting up the windows, automatically running e.g. **rthreadsJoin** in each one, would be desirable. The above utilities can accomplish this by enabling us to have code running in one window write a specified string to another window. E.g. say we are in the shell of Window A. We can do, e.g.

```
screen -S WindowBScreen
```

in Window B to start a **screen** session there. Then in Window A we might run

```
screen -S WindowB -X stuff 'ls'$'\n'
```

and the Unix **ls** command will run in Window B just as if we had typed it there ourself! Moreover, we might be running R in Window A, in which case we can run the above shell command via R's **system** function

In other words, we can for instance automate the running of **rthreadsJoin** in all the windows, instead of having to type the command in each one.

## To Learn More

- N. Matloff, *Parallel Computing for Data Science With Examples in R, C++ and CUDA*
- Tutorial on accessing OpenMP via **Rcpp**

## Legal