Parallel R

We will explore how to optimize parallelization to efficiently run on TACC resources. The key to this process is understanding the resource requirements of your workflow, particularly those sections that dominate the time to completion. Often times, these sections of code are embarrassingly parallel and can thus be split into simultaneous execution.

myProc()

To begin, let's assume we are working on a system with 4 cores and 8GB of memory. Next, we'll create a function representing the core 'work' for our task. We'll want to run this proc 100x and then take the avg of the results.

```r
myProc <- function(size=10000000) {
  # Load a large vector
  vec <- rnorm(size)

  # Now sleep on it
  Sys.sleep(2)

  # Now sum the vec values
  return(sum(vec))
}
```

Profile

In order to gain an understanding of the resources requirements this functions takes, we can run some profiling.

```r
ptm <- proc.time()
myProc(100000000)

## [1] -4195.251

proc.time() - ptm

## user   system elapsed
## 8.123   0.287   10.422

Sys.getpid()

## [1] 69542

system('top -b -n 1 -u $USER', intern=TRUE)

## [1] "top - 10:59:57 up 8 days, 2:35, 0 users, load average: 0.49, 2.08, 8.08"
## [2] "Tasks: 494 total, 1 running, 493 sleeping, 0 stopped, 0 zombie"
## [3] "Cpu(s): 56.4%us, 1.0%sy, 0.0%ni, 42.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st"
## [4] "Mem: 32815324k total, 6622108k used, 26193216k free, 11648k buffers"
## [5] "Swap: 4194296k total, 0k used, 4194296k free, 1009920k cached"
## [6] ""
## [7] " PID USER      PR NI VIRT  RES  SHR S %CPU %MEM   TIME+ COMMAND      
## [8] 69552 rhuang   20  0 15300 1456  844 R   2.0   0.0   0:00.02 top"
```
Plan

For the size=100M(10M), we see that this proc takes:
~ 10(3) secs
~ 800(100)MB of ram
These numbers can vary greatly based on the hardware and how R was compiled.
Notice only a single R process in the output from 'top'

Serial - for loop

Let's first create a serial version of R code to run myProc() 10x.
```r
ptm <- proc.time()
result <- c()
for(i in 1:10) {
  result <- c(result, myProc())
}
proc.time() - ptm
```
```
##    user  system elapsed
##  8.143  0.333  28.503
```

Serial - apply

Now let's see if we can improve things by using functions from the apply family.
```r
ptm <- proc.time()
result <- sapply(1:10, function(i) myProc())
proc.time() - ptm
```
```
##    user  system elapsed
##  8.111  0.296  28.434
```
Sys.getpid()

## [1] 69542

system('top -b -n 1 -u $USER', intern=TRUE)

## [1] "top - 11:00:55 up 8 days, 2:36, 0 users, load average: 0.37, 1.76, 7.60"
## [2] "Tasks: 494 total, 1 running, 493 sleeping, 0 stopped, 0 zombie"
## [3] "Cpu(s): 56.4%us, 1.0%sy, 0.0%ni, 42.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st"
## [4] "Mem: 32815324k total, 6387496k used, 26427828k free, 11752k buffers"
## [5] "Swap: 4194296k total, 0k used, 4194296k free, 1010196k cached"
## [6] "
## [7] " PID USER      PR NI VIRT  RES  SHR S %CPU %MEM    TIME+ COMMAND 
## [8]  69376 rhuang   20  0 105m  1428 1072 S   0.0   0.0    0:00.01 slurm_script
## [9]  69478 rhuang   20  0 105m   720  360 S   0.0   0.0    0:00.00 slurm_script
## [10]  69479 rhuang   20  0  287m  6756 5696 S   0.0   0.0    0:00.18 rserver
## [11]  69493 rhuang   20  0  60592 1072  244 S   0.0   0.0    0:00.00 ssh
## [12]  69495 rhuang   20  0  60592 1076  244 S   0.0   0.0    0:00.00 ssh
## [13]  69497 rhuang   20  0  60592 1072  244 S   0.0   0.0    0:00.00 ssh
## [14]  69513 rhuang   20  0 105m   720  360 S   0.0   0.0    0:00.00 slurm_script
## [15]  69514 rhuang   20  0  98.5m  584  508 S   0.0   0.0    0:00.00 sleep
## [16]  69527 rhuang   20  0 1574m  954m 18m S   0.0   3.0    0:05.87 rsession
## [17]  69542 rhuang   20  0  847m  576m 8996 S   0.0   1.8    0:25.73 R
## [18]  69564 rhuang   20  0 15300 1452  844 R   0.0   0.0    0:00.01 top
## [19]  69565 rhuang   20  0  98.5m  584  508 S   0.0   0.0    0:00.00 sleep
## [20] "

**Parallel - Plan**

To run 4 procs in parallel, we'd need 4 * 800MB = 3.2 GB.
If we had 16 cores, we'd need 16 * 800MB = 12.8 * GB. This is larger than we have on our hypothetical machine.
At most, you'd want to 'require' < 70% of the entire nodes memory. This is a rule of thumb and could be tweaked for each problem.
In the 16 core case, we'd want to only use 8GB * .7 = 5.6GB -> 5.6GB / .8GB = 7 cores.

**library(parallel): mclapply**

The parallel package provides several methods for parallelizing your work.
Perhaps the easiest to get started with is 'mclapply'.
This function utilizes 'forking' to spin up additional R processes and is thus only available on Linux-like resources.
'mclapply' can only be used for single-node parallelism.

```
library(parallel): mclapply
```

```
require(parallel)
```

```
# Loading required package: parallel
```

```
ptm <- proc.time()
result <- mclapply(1:10, function(i) myProc(), mc.cores=4)
proc.time() - ptm
```

```
# user system elapsed
# 0.004 0.023 8.550
```

```
Sys.getpid()
```

```
## [1] 69542
```

```
system('top -b -n 1 -u $USER', intern=TRUE)
```

```
## [1] "top - 11:01:04 up 8 days, 2:36, 0 users, load average: 0.34, 1.73, 7.55"
## [2] "Tasks: 498 total, 1 running, 497 sleeping, 0 stopped, 0 zombie"
## [3] "Cpu(s): 56.4%us, 1.0%sy, 0.0%ni, 42.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st"
## [4] "Mem: 32815324k total, 6576400k used, 26238924k free, 11764k buffers"
## [5] "Swap: 4194296k total, 0k used, 4194296k free, 1010236k cached"
## [6] "PID USER      PR NI    VIRT   RES   SHR S %CPU %MEM    TIME+ COMMAND
## [7] 69376 rhuang   20  0 105m  1428  1072 S  0.0  0.0  0:00.01 slurm_script
## [8] 69478 rhuang   20  0  105m   720   360 S  0.0  0.0  0:00.00 slurm_script
## [9] 69479 rhuang   20  0  287m  6756  5696 S  0.0  0.0  0:00.18 rserver
## [10] 69493 rhuang   20  0  60592  1072  244 S  0.0  0.0  0:00.00 ssh
## [11] 69495 rhuang   20  0  60592  1076  244 S  0.0  0.0  0:00.00 ssh
## [12] 69497 rhuang   20  0  60592  1072  244 S  0.0  0.0  0:00.00 ssh
## [13] 69513 rhuang   20  0  105m   720   360 S  0.0  0.0  0:00.00 slurm_script
## [14] 69514 rhuang   20  0  98.5m   584   508 S  0.0  0.0  0:00.00 sleep
## [15] 69527 rhuang   20  0 1574m  954m  18m S  0.0  3.0  0:05.87 rsession
## [16] 69542 rhuang   20  0  549m  276m   924 S  0.0  0.9  0:25.83 R
## [17] 69565 rhuang   20  0  98.5m   584   508 S  0.0  0.0  0:00.00 sleep
## [18] 69566 rhuang   20  0  701m  420m  1484 S  0.0  1.3  0:02.52 R
## [19] 69567 rhuang   20  0  701m  420m  1484 S  0.0  1.3  0:02.51 R
## [20] 69568 rhuang   20  0  625m  344m  1484 S  0.0  1.1  0:01.68 R
## [21] 69569 rhuang   20  0  625m  344m  1484 S  0.0  1.1  0:01.67 R
```
library(parallel): snow

SNOW uses either MPI or socket based connections to achieve parallelism. This means it can use cores on both the local and remote nodes. There is a bit more work involved with setting up the SNOW 'cluster'. Most of the 'snow' functionality was added to the 'parallel' package. However, more fine grained control is achieved with using the 'snow' package directly. With SNOW, you must explicitly make vars/functions available in the sub-processes.

library(parallel): snow

SNOW single node example

```r
require(snow)

# Loading required package: snow

# Attaching package: 'snow'

# The following objects are masked from 'package:parallel':

# clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
# clusterExport, clusterMap, clusterSplit, makeCluster,
# parApply, parCapply, parLapply, parRapply, parSapply,
# splitIndices, stopCluster

hostnames <- rep('localhost', 4)
cluster <- makeSOCKcluster(hostnames)

clusterExport(cluster, list('myProc'))

ptm <- proc.time()
result <- clusterApply(cluster, 1:10, function(i) myProc())
proc.time() - ptm

# user  system elapsed
# 0.004  0.002  8.632

stopCluster(cluster)
```

library(parallel): snow

SNOW will launch a new R process on each of the inputs to hostnames. If we pass in 4 values of 'localhost', this is equivalent to using 4 cores on localhost. When sending work to multiple nodes, there are a number of options for using all cores on all nodes. These options provide great flexibility for utilizing all cores on all nodes efficiently, but take extra work.

library(parallel): foreach

'foreach' is a package that makes parallelization easier.
'foreach' uses other parallel functions under the hood, but hides some of the complexity.

### library(parallel): foreach + multicore

```r
library(parallel)
require(foreach)
## Loading required package: foreach
require(doMC)
## Loading required package: doMC
## Loading required package: iterators

registerDoMC(cores=4)

ptm <- proc.time()
result <- foreach(i = 1:10, .combine=c) %dopar% {
  myProc()
}
proc.time() - ptm
## user    system elapsed
## 0.024    0.014    8.570
```

### library(parallel): foreach + snow

Notice that 'foreach' handles the clusterExport for us.

```r
library(parallel)
require(foreach)
## Loading required package: foreach
require(doSNOW)
## Loading required package: doSNOW

hostnames <- rep('localhost', 4)
cluster <- makeSOCKcluster(hostnames)
registerDoSNOW(cluster)

ptm <- proc.time()
result <- foreach(i = 1:10, .combine=c) %dopar% {
  myProc()
}
proc.time() - ptm
## user    system elapsed
## 0.021    0.004    8.697

stopCluster(cluster)
```

### library(parallel): multi-node

You could use SNOW and manually manipulate the 'hostnames' parameter to achieve multinode. This currently runs into problems with having your environment set correctly for R with TACC modules.

```r
library(parallel)
require(foreach)
require(doSNOW)
```
# Get backend hostnames
hostnames <- scan("nodelist.txt", what="", sep="\n")

# Set reps to match core count'
num.cores <- 4
hostnames <- rep(hostnames, each=num.cores)
hostnames

cluster <- makeSOCKcluster(hostnames)
registerDoSNOW(cluster)

ptm <- proc.time()
result <- foreach(i=1:10, .combine=c) %dopar% {
  myProc()
}
proc.time() - ptm

stopCluster(cluster)

**RMPISNOW**

An easier mechanism is to launch your Rscript via ibrun + RMPISNOW

```bash
#!/bin/bash
#SBATCH -J parallel_lab       # Job name
#SBATCH -o parallel_lab_jobout.o%j # Name of stdout output file (%j expands to jobId)
#SBATCH -e parallel_lab_joberr.o%j # Name of stderr output file(%j expands to jobId)
#SBATCH -n 40                  # Total number of mpi tasks requested
#SBATCH -p vis                 # Submit to the 'normal' or 'development' queue
#SBATCH -t 00:30:00            # Run time (hh:mm:ss)
#SBATCH -A Sept2014DataAnalysis # Allocation name to charge job against

#Set environment
module purge
module load TACC
module load Rstats

cd /home/00157/walling/training/parallel

# call R code vis RMPISNOW
# RMPISNOW handles setting up the cluster so each core is utilized
ibrun RMPISNOW < ./MultiNodeMultiCore.R
```

**SLURM Parameters**

The variable: `SBATCH -n X` specifies the number of cores to use
The variable: `SBATCH -N X` specifies how many nodes to use.
If we know the memory requirements are such that I can only use half the cores(10) on a given node, and I want a total of 100 cores...
...then for Maverick, with 20 cores/node, we could use:

`#SBATCH -N 10`
`#SBATCH -n 100`
If we specify only `SBATCH -n 100`, we'd get 5 nodes on Maverick and every core would run an R process.