A tutorial on slurm system and rslurm

Javad Rahimikollu,Kayhan Batmanghelich

May 15, 2018, 14:01

Contents

Introduction

Simple Linux Utility for Resource Management(Slurm) is an open source job scheduling system for large and small Linux clusters. Slurm has three key functions:

- It allows access to resources (compute nodes) based on certain criteria and parametrs for users to perform computations.
- It ensbles users with a framework for starting, executing, and monitoring work on a group of assigned compute nodes.
- It manages high number of access requests to compute nodes by assigning them in a queue.

In slurm cluster systems, user first sign in to the head node and using and request access to computing nodes based on job parameters. Here is the list of some of these parameters:

```
Slurm_Options<-read.table("Slurm_Options.csv",header = TRUE, sep=",")
pander(Slurm_Options)
```

option	Description
-j	jobname
-t	Walltime requested HH:MM:SS
-p	Partition
-N	Number of Nodes
-A	Account or Group Name
–mem	Requeted Memory in GB
–ntasks-per-node	Number of cores to allocate per node

Slurm Architecture

The slurm consits of head nodes and compute nodes. The head nodes are the nodes user signs in and is not meant for any computation. The head node is concted to slurm job distribution system. When user request a

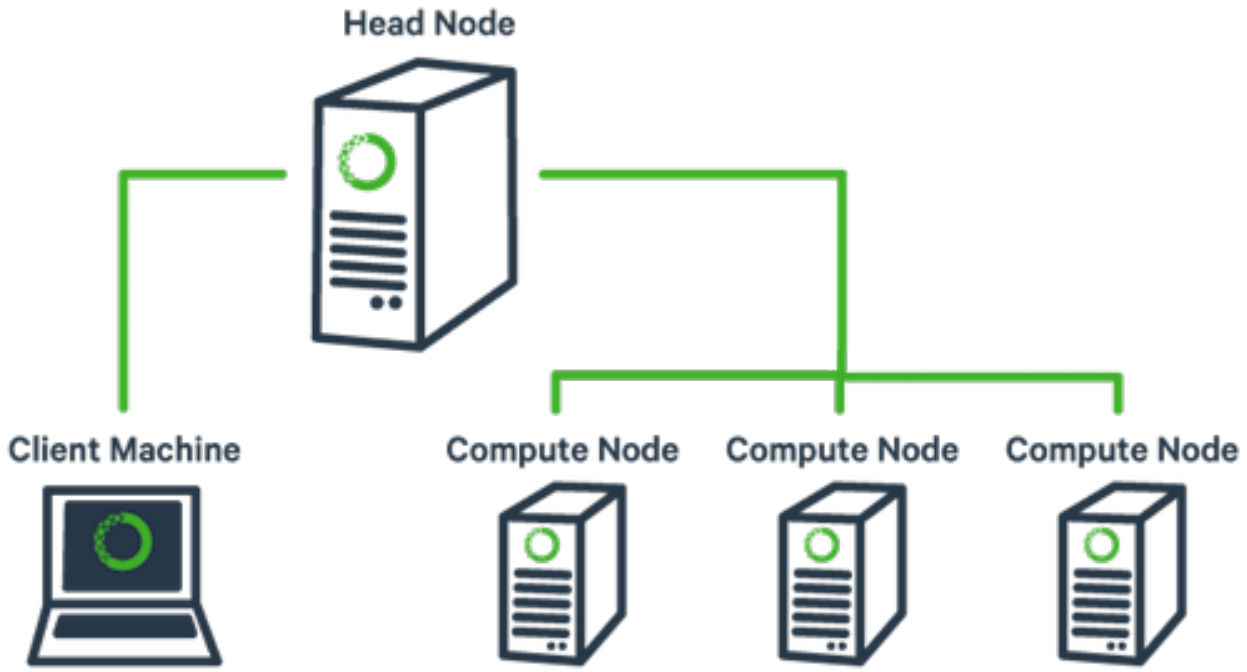

Figure 1: The slurm structure

node from compute nodes, slurm looks for available compute nodes based on the paramters of the job. If the compute node is available a compute node will be assigned other wise the job will be in the queue.

Slurm commands

sbatch: To submit the slurm job, one can create a bash script with all the paramters and use **sbatch** command to submit the job. Here is an example of a batch script. If we name this bash scipt as test.sh, we can use sbatch test.sh to submit this job to run Test.R. Here is the example of a very simple batch script which runs the Test.R rscript. Test.R only print *Hi slurm*.

This is the rscript Test.R

```
print("Hi slurm")
```

```
## [1] "Hi slurm"
```

```
!/bin/bash
#SBATCH - N 1
#SBATCH -t 00:20:00
#SBATCH -J my_job
#SBATCH --mem 10GB

Rscript ~/Test.R
```

```
sbatch submit_test.sh
```

submits the *submit.sh* bash script.

squeue: When the user wants to see the status of the job and whether it is in the queue or it in the compute node.

squeue -u javad

We want to see the status jobs for user javad

scancel: When the user want to cancel a job

scancel 1345

we want to cancel the job with id of 1345

Rslurm

Lets assume, we want to submit a rscript with iterative argument. Using the bash script we have to submit one job for each setting of argument which will need bash programming to enumerate all the values of argument. The rslurm package facilitates the process of distributing this type of calculation over computing nodes in the Slurm workload manager. The main function, slurm_apply, automatically splits the computation across multiple nodes and writes the submission scripts. It also includes functions to retrieve and combine the output from different nodes, as well as wrappers for common Slurm commands.

Basic Example

Lets say our task is to find $E(x^2 sin(x * \pi))$ where $x \sim N(0, 1)$.

The naive way The super naive way of doing this is to create an r script and do all the sampling and submit the job. This is naive becuase we are doing all the calculations in one core. Lets assume we want to draw 10^6 sample and calculte the expected value. This script is created as *naive_Expected_value*.

```
start<-Sys.time()
samp <- rnorm(10^6, 0, 1)

E<-samp^2*sin(samp*pi/8)

mean(E)
```

```
## [1] -0.0003285379
```

```
var(E)
```

```
## [1] 1.619812
```

```
stop<-Sys.time()

stop-start
```

```
## Time difference of 0.3261874 secs
```

Lets do the process for 10^5 samples:

```
start<-Sys.time()
samp <- rnorm(10^5, 0, 1)

E<-samp^2*sin(samp*pi/8)

mean(E)
```

```
## [1] -0.001302753
```

```
var(E)
```

```
## [1] 1.595019
```

```
stop<-Sys.time()

stop-start
```

```
## Time difference of 0.0325315 secs
```

We see that the time difference is considerable. *Using Bash Scripts*:The result above, leads us to the next solution, we can crate an executable rscript and run 10 of 10^5 sample in parallel in defferent cpus. The execuatable rscript can be created as follows:

```
#!/usr/bin/env Rscript
library(docopt)
```

```
## Warning: package 'docopt' was built under R version 3.4.1
```

```
'Usage:
   Expected_value_chuncks.R [-i <i>]

Options:
   -i Chunck Number [default: 1]

 ]' -> doc


opts <- docopt(doc)

i<-opts$i


samp <- rnorm(10^5, 0, 1)

E<-samp^2*sin(samp*pi/8)

mean(E)
```

```
## [1] 0.0005324527
```

```
var(E)
```

```
## [1] 1.639685
```

```
file_name<-paste0("chunk_",i,".csv")
write.table(cbind(i,mean(E)),file=file_name)
```

The *#!/usr/bin/env Rscript* is an indicator of executable rscript for the linux system. The argument i is the chunck number of the sampling. The mean of sample from chunk number i will be written in to a file name *chunk_i.csv* we can run this r script in linux environment as follows:

```
./Expected_value_chuncks.R 1
```

```
## WARNING: ignoring environment value of R_HOME
## Loading required package: methods
## [1] 0.007337291
```

Inorder to submit this job interavtively we can write the sbatch script as follows:

```
#!/bin/bash
#SBATCH -t 1:00:00
#SBATCH --job-name="chunck${$1}"
#SBATCH --output="chunck${$1}".out
#SBATCH -n 2
#SBATCH --mem=10GB
echo "SLURM_JOBID="\\$SLURM_JOBID
echo "SLURM_JOB_NODELIST"=\\$SLURM_JOB_NODELIST
echo "SLURM_NNODES"=\\$SLURM_NNODES
echo "SLURMTMPDIR="\\$SLURMTMPDIR
echo "working directory = /pghbio/dbmi/batmanlab/javad/MultiVariate/Bash/ "
echo "$(ls)"

STARTTIME=$(date +%s)

ulimit -s unlimited
# The initial srun will trigger the SLURM prologue on the compute nodes.
echo "Launch script for javad@pitt.edu"
module load R/3.4.1-mkl

Rscript ./Expected_value_chuncks $1




ENDTIME=$(date +%s)
ELAPSED=$(($ENDTIME - $STARTTIME))

echo "Execution duration: $((ELAPSED/3600))h:$(((ELAPSED/60)%60))m:$((ELAPSED%60))s"
echo "Usage statistics for [$SLURM_JOB_ID]:"


echo "All Done!"
```

This is an executable bash script which takes the first argument *$1* as the chunck number and submit the job. Inodered to submit this job iteratively we should have anothe bash script with a for loop to iteratively submit the job.

```
#!/bin/bash

for i in {1..10}
do
sbatch Single_Job.sh $i
done

echo "All Done!"
```

```
## Submitted batch job 2995311
## Submitted batch job 2995312
```

```
## Submitted batch job 2995313
## Submitted batch job 2995314
## Submitted batch job 2995315
## Submitted batch job 2995316
## Submitted batch job 2995317
## Submitted batch job 2995318
## Submitted batch job 2995319
## Submitted batch job 2995320
## All Done!
```

Now lets read the *.csv* files from each chunck.

```
data1<-read.csv(file ="chunk_1.csv",stringsAsFactors=FALSE,sep="")
data2<-read.csv(file ="chunk_2.csv",stringsAsFactors=FALSE,sep="")
data3<-read.csv(file ="chunk_3.csv",stringsAsFactors=FALSE,sep="")
data4<-read.csv(file ="chunk_4.csv",stringsAsFactors=FALSE,sep="")
data5<-read.csv(file ="chunk_5.csv",stringsAsFactors=FALSE,sep="")
data6<-read.csv(file ="chunk_6.csv",stringsAsFactors=FALSE,sep="")
data7<-read.csv(file ="chunk_7.csv",stringsAsFactors=FALSE,sep="")
data8<-read.csv(file ="chunk_8.csv",stringsAsFactors=FALSE,sep="")
data9<-read.csv(file ="chunk_9.csv",stringsAsFactors=FALSE,sep="")
data10<-read.csv(file ="chunk_10.csv",stringsAsFactors=FALSE,sep="")


Data<-rbind(data1,data2,data3,data4,data5,data6,data7,data8,data9,data10)


print(Data)
```

```
##                i
## 1   0.0073372910
## 2   0.0011812429
## 3   0.0034889589
## 4   0.0029420581
## 5  -0.0041356177
## 6   0.0057934831
## 7  -0.0013952110
## 8  -0.0091144704
## 9  -0.0005805678
## 10 -0.0042799460
```

We can get the mean of the 10 chuncks as approximation for expected value.

```
mean(Data$i)
```

```
## [1] 0.0001237221
```

Now, lets use rslurm for this problem, the equvalent of rscript *Expected_value_chunks.R* is a function *test_func*.

```
test_func <- function(chunk_num) {
    .libPaths(c("/home/javad/R/x86_64-pc-linux-gnu-library/3.4","/opt/packages/R/3.4.1-mkl/lib64/R/libra
    samp <- rnorm(10^5, 0, 1)
    E<-samp^2*sin(samp*pi/8)
    mean(E)

}
```

To pass the different values of arguments *chunk_num* we store them in a dataframe.

```
pars <- data.frame(chunk_num = 1:10)
head(pars, 3)
```

```
##   chunk_num
## 1         1
## 2         2
## 3         3
```

Now, we can treat the *test_func* as the rscript which will use the *pars dataframe* and submit the job for each row *pars dataframe* using *slurm_apply*.

```
library(rslurm)
```

```
## Warning: package 'rslurm' was built under R version 3.4.1
```

```
sjob <- slurm_apply(test_func, pars, jobname = 'test_apply',nodes = 10, cpus_per_node = 2, submit = TRU
```

on the background, slurm_apply will create a submit script with the job options provided in the function argument.i.e nodes and etc. Folders with the name of *results_node_indicator.RDS* file for each node. As we can see below there are two folders with the name of *results_0.RDS* and *results_1.RDS* for *N=2* nodes.

Now, we can use *get_slurm_out* to get the output for the each row of *par* dataframe on the *test_func* function. As we can see from *res* data.frame we have 10 rows, each row for each run.

```
res <- get_slurm_out(sjob, outtype = 'table')
res
```

```
##                V1
## 1   0.0042051496
## 2   0.0083953871
## 3  -0.0003217620
## 4  -0.0073527301
## 5   0.0071013611
## 6  -0.0071373804
## 7  -0.0024928521
## 8  -0.0078402191
## 9  -0.0007974452
## 10  0.0012001005
```

So we can get the estimated expected value from *rslurm* results.

```
mean(res$V1)
```

```
## [1] -0.0005040391
```

using *rslurm*, we avoid using the the extra bash files to submit the job.

Machine Learning Example

Support vector machines are one of the techniques which widely used in machine learning. Lets assume that we want to find the hyperamters of RBF kernels,σ and C. In this section we want to performa a grid search using rslurm. we are going to use *iris* data to fit *Sepal.Width* from *Sepal.length*.

First lets define the *par* data frame for this problem.

```
pars <- data.frame(par_cost = seq(0.1,1,0.1),
                   par_sigma = seq(0.1, 1,0.1))
head(pars, 3)
```

```
##   par_cost par_sigma
## 1      0.1       0.1
## 2      0.2       0.2
## 3      0.3       0.3
```

Now lets define our main function to fit the data:

```
svm_func<-function(par_cost,par_sigma){
set.seed(7);
library(datasets)
library(e1071)
data("iris")
  #Randomly shuffle the data
iris<-iris[sample(nrow(iris)),]
#Create 10 equally size folds
folds <- cut(seq(1,nrow(iris)),breaks=10,labels=FALSE)
#Perform 10 fold cross validation
err<-numeric()
for(i in 1:10){
  #Segement your data by fold using the which() function
  testIndexes <- which(folds==i,arr.ind=TRUE)
  testData <- iris[testIndexes, ]
  trainData <- iris[-testIndexes, ]
  #Use the test and train data partitions however you desire...
  svp <- ksvm(trainData$ Sepal.Length,trainData$Sepal.Width,C=par_cost,nu=0.2,kpar=list(sigma=par_sigma
  err[i]=sum((testData$Sepal.Width-predict(svp,testData$Sepal.Length))^2)
}

  mean(err)

}
```

Rslurm has module issues, to overcome that problem we can write the slurm_apply function without submiting it. In the _rslurm_job_object, add the module into the *submit.sh* script.

```
library(rslurm)
SVM <- slurm_apply(svm_func, pars, jobname = 'SVM',nodes = 10, cpus_per_node = 2, submit = FALSE)
```

```
## Submission scripts output in directory _rslurm_SVM
```

For example for the SVM job, we went to the folder *_rslurm_SVM* and add the **module load R/3.4.1-mkl** into the *submit.sh*

```
res <- get_slurm_out(SVM, outtype = 'table')
cbind(res,pars)
```

```
##          V1 par_cost par_sigma
## 1  2.820763      0.1       0.1
## 2  2.890028      0.2       0.2
## 3  2.828655      0.3       0.3
## 4  2.749627      0.4       0.4
## 5  2.740535      0.5       0.5
## 6  2.700828      0.6       0.6
## 7  2.694224      0.7       0.7
## 8  2.683297      0.8       0.8
## 9  2.695140      0.9       0.9
## 10 2.700858      1.0       1.0
```

Practice problem

We have seen the hyperparameter optimization for values of c and σ. Can you perform the same concept on the blocks of C and σ?(each job gets a block of *sigma* and c to explore).**hint: the block number is the function argument**