An example of using JASMIN with R

Tom August

20 February, 2018

Contents

Introduction

This document and the accompanying code as designed to give you an introduction to show you how to use JASMIN to parallelise your R code. We will start with some R code and work though how we can make it parallel. We will then go through how we can get it to run on JASMIN.

Modelling the distribution of 'settlers'

In our example code we are exploring the way in which 'settlers' distribute themselves around established 'pioneers'. Think of this as a simulation of the American gold-rush.

We take an area (it could be a country, state or city) that is made up of units (cells) that can be settled (occupied). We allow the initial 'pioneers' to settle first (seeding the cells). We then establish some rules for future residents to settle (settling rules). Once we have run the simulation we plot the area, and collect some stats to study the growth of the population under different starting conditions and different settling rules.

There are a range of different parameters that we can change, and each will affect the results. We want to study the effect of four parameters:

- kNumSettlers – The number of settlers who follow the pioneers
- kMaxLookAroundSteps - How many scouting attempts will settlers make, before abandoning
- seeding.opt – How are pioneers seeded? 1: Random, 2: Central Square, 3: Central Ring, 4: Two Columns.
- settling.option - How are settlers seeded? 1: Random, 2: NSEW only, 3: Diagonal settling only. Run the script and see what it produces. Try changing some of the parameter values above and see what changes.

This code was originally written by Matt Asher for statisticsblog.com

```
# First we need to install a few packages
install.packages('ggplot2')
install.packages('plyr')
install.packages('reshape')
```

```
library(ggplot2)
library(plyr)
library(reshape)
```

```
##
## Attaching package: 'reshape'
```

```
## The following objects are masked from 'package:plyr':
##
##     rename, round_any
```

```
# Make sure you are in the working directory containing the
# example code and then run
source("scripts/simFunctions.R")
```

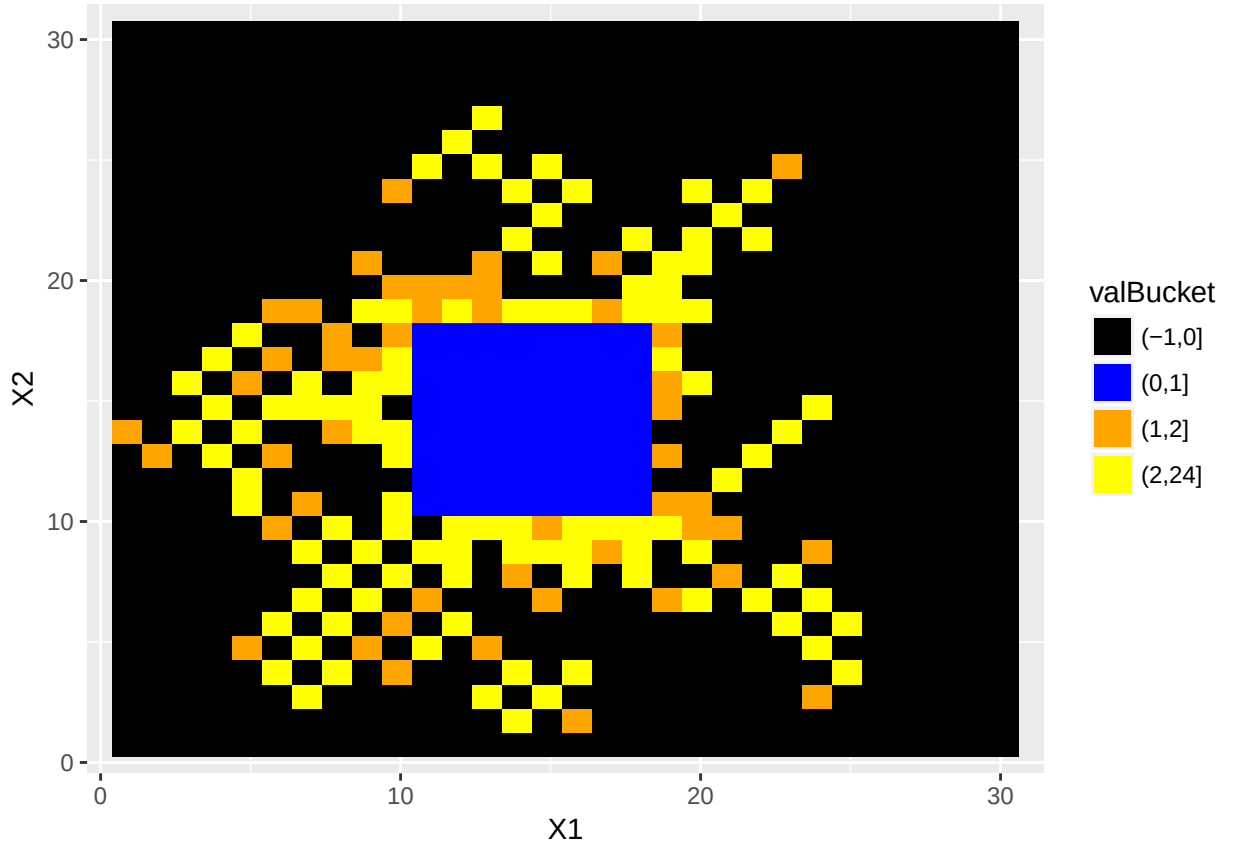
Now we have the additional functions loaded we can run the main part of the script which sets up various parameters and then runs 10 simulations, averaging the results across them

```
kNumReplications <- 10

# Size of the area
areaW <- 30
areaH <- 30

# Homesteaders, they don't care about finding a neighbor
# only used in random seeding
numPioneers <- 30

#These are the people who follow the pioneers
kNumSettlers <- 500
# How many scouting attempts will settlers make, before abandoning
kMaxLookAroundSteps <- 20

#Try out different seeding rules
# (1:Random) (2: Central Square) (3:Central Ring) ( 4: Two Columns)
seeding.opt <- 2
# (1: Random) (2:NEWS only) (3:Diagonal settling only)
settling.option <- 3

adjacells <- getAdjacentCellsDataFrame()

# Make multiple runs (Replication of simulation) and take the average of stats
st <- data.frame()
st_row<- vector()
for(i in 1:kNumReplications) {
  area_df <- resetIteration()

  seedAreaWithPioneers(numPioneers,seeding.opt)
  simstats <- accommodateSettlers(kNumSettlers, settling.option)

  found.home <- simstats[1]
  max.look.around <- simstats[2]
  #compute for this iterations
  st_row <- store_iteration_stats(i, kNumSettlers, found.home, max.look.around)
  st <- rbind(st,st_row)
```

```
}
```

```
## 110 settled out of 500 22%
## Number of Settlers who hit max look around: 390
##
## 109 settled out of 500 22%
## Number of Settlers who hit max look around: 391
##
## 93 settled out of 500 19%
## Number of Settlers who hit max look around: 407
##
## 148 settled out of 500 30%
## Number of Settlers who hit max look around: 353
##
## 131 settled out of 500 26%
## Number of Settlers who hit max look around: 369
##
## 166 settled out of 500 33%
## Number of Settlers who hit max look around: 334
##
## 136 settled out of 500 27%
## Number of Settlers who hit max look around: 365
##
## 155 settled out of 500 31%
## Number of Settlers who hit max look around: 345
##
## 141 settled out of 500 28%
## Number of Settlers who hit max look around: 360
##
## 151 settled out of 500 30%
## Number of Settlers who hit max look around: 349
```

```
#Render
p <- drawArea(area_df)
p
```



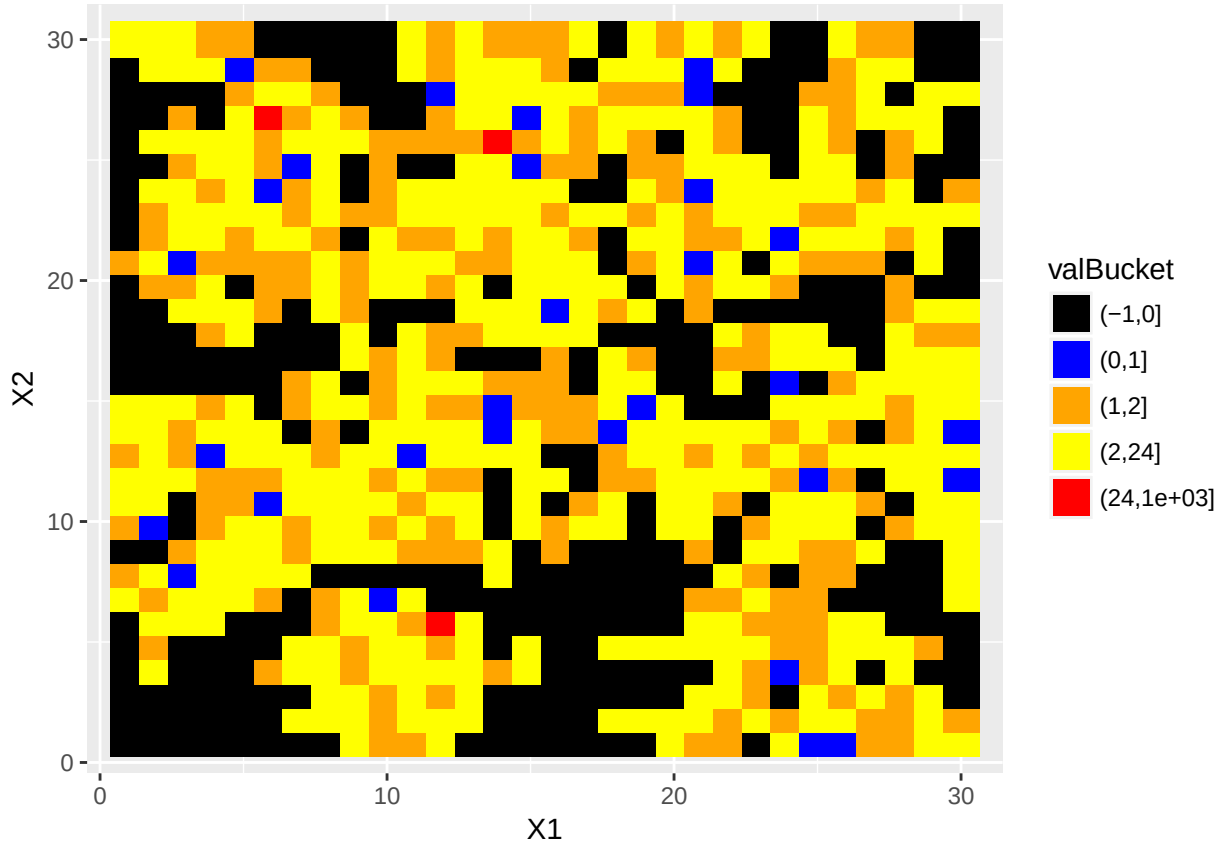

```
names(st) <- c("Iter", "FoundHome", "NumSettlers", "Percent")
st
```

```
##    Iter FoundHome NumSettlers Percent
## 1     1       110         500    22.0
## 2     2       109         500    21.8
## 3     3        93         500    18.6
## 4     4       148         500    29.6
## 5     5       131         500    26.2
## 6     6       166         500    33.2
## 7     7       136         500    27.2
## 8     8       155         500    31.0
## 9     9       141         500    28.2
## 10   10       151         500    30.2
```

Run the script and see if you can replicate these results (this is available in `original_script.R`) . Try changing some of the parameter values above to these values and see what changes:

- Set `seeding.opt` to 1
- Set `settling.option` to 2
- Set `kMaxLookAroundSteps` to 30
- Set `kNumSettlers` to 750

If you get things set up correctly you should see something like this



Scaling up our analysis

Having completed this work we have now been asked to run these simulations for a large range of parameter values so that we can examine the sensitivity of the system to changes in the parameters. To do this we have been given a file which contains all of the parameter combinations that we need to test.

```
# Lets load in the parameters file and have a look
parameters <- read.csv('data/parameters.csv')

head(parameters)
```

```
##   kMaxLookAroundSteps seeding.opt settling.option kNumSettlers
## 1                  10           1               1          100
## 2                  20           1               1          100
## 3                  30           1               1          100
## 4                  40           1               1          100
## 5                  50           1               1          100
## 6                  10           2               1          100
```

```
nrow(parameters)
```

```
## [1] 300
```

Each row of this files gives a different set of parameter combinations for our code to work with. You could also have this as a list of paths to data files, species names, lake co-ordinates, etc. This parameter file can be used in a range of ways to allow you to run lots of code in parallel, more later.

For now let's think about how we can change our R script so that it works with this `.csv` file to loop through all the model combinations. *Have a go at doing this yourself before you read the solution below.*

```
kNumReplications <- 5

# Size of the area
areaW <- 30
areaH <- 30

# Homesteaders, they don't care about finding a neighbor
# only used in random seeding
numPioneers <- 30

parameters <- read.csv('data/parameters.csv')

# In testing I only want to run the loop for the
# top few rows of the parameters file
parameters <- head(parameters)

for(i in (1:nrow(parameters))){

  cat('\n\nStarting parameter set', i, '\n')

  #These are the people who follow the pioneers
  kNumSettlers <- parameters[i, 'kNumSettlers']

  # How many scouting attempts will settlers make, before abandoning
  kMaxLookAroundSteps <- parameters[i, 'kMaxLookAroundSteps']

  #Try out different seeding rules
  # (1:Random) (2: Central Square) (3:Central Ring) ( 4: Two Columns)
  seeding.opt <- parameters[i, 'seeding.opt']

  # (1: Random) (2:NEWS only) (3:Diagonal settling only)
  settling.option <- parameters[i, 'settling.option']

  adjacells <- getAdjacentCellsDataFrame()

  # Make multiple runs (Replication of simulation) and take the average of stats
  st <- data.frame()
  st_row <- vector()
  for(i in 1:kNumReplications) {
    area_df <- resetIteration()

    seedAreaWithPioneers(numPioneers,seeding.opt)
    simstats <- accommodateSettlers(kNumSettlers, settling.option)

    found.home <- simstats[1]
    max.look.around <- simstats[2]
    #compute for this iterations
    st_row <- store_iteration_stats(i, kNumSettlers, found.home, max.look.around)
    st <- rbind(st,st_row)
  }
```

```
  # Render plot
  plotname <- paste0('Pop_map_',
                     paste(kNumSettlers, kMaxLookAroundSteps, seeding.opt, settling.option, sep = '_'),
                     '.png')
  png(filename = file.path('output_figures', plotname),
      width = 6, height = 4, units = 'in', res = 300)
    drawArea(area_df)
  dev.off()
  names(st) <- c("Iter", "FoundHome", "NumSettlers", "Percent")
  tablename <- paste0('Pop_table_',
                      paste(kNumSettlers, kMaxLookAroundSteps, seeding.opt, settling.option, sep = '_')
                     '.csv')
  write.csv(st, file = file.path('output_tables', tablename), row.names = FALSE)
}
```

You will see that this code is very similar to the code we had before (available in `loop_parameters_file.R`). We have changed the setup of the parameter values so that they are read in from the `.csv` file that has been provided. The loop uses an index, `i`, to pull out the correct parameters from the parameters table. We have also changed the script so that the plots and the table of results are written out to two folders, one for plots and one for tables. This setup could now be run across all the rows in our parameter files. This example takes ~30 minutes to run locally, but could be much longer if we increased the number of iterations or the number of parameters we varied.

- Is a 30 minute job long enough to warrant putting on JASMIN?
- How could this same approach be used with an example of your own work?
- Could this have been done in a better way?

Move files to JASMIN

You should by now have set yourself up a JASMIN account and be familiar with using MobaXterm to access JASMIN and transfer files. Now use MobaXterm to move all of the files and scripts you have been provided with to your home space of JASMIN. Once you have done this use `cd` to change your working directory so that you are inside the top directory. Once you have done this, typing `ls` should reveal all the files and folders in your working directory which will include the files `original_scrip.R`, `loop_parameters_file.R` and others. If you don't see these you have either not copied across all the files correctly or you are not in the correct directory.

Creating a job file

The job file is vital to running your code on JASMIN, it has all the important information such as which queue you job will go to, how much time you need, etc. Here is the job script that I have created for this job:

```
#!/bin/bash
#BSUB -q short-serial
#BSUB -n 1
#BSUB -W 10:00
#BSUB -J Pop_Sim[1-30]
#BSUB -oo R-%J-%I.o
#BSUB -eo R-%J-%I.e
R CMD BATCH "--args ${LSB_JOBINDEX}" JASMIN_parameters_file.R "console/console${LSB_JOBINDEX}.Rout"
```

Let's go through this line by line so that you know what it means NOTE: I have reduced the number of parameter sets from 300 to 30 so that we don't swamp JASMIN as a group!

`#!/bin/bash` - this says the file is a bash script, this tells Linux how the file should be run. `#BSUB` - All lines that start with this are instructions to the queue system `-q short-serial` - Submit this job to the `short-serial` queue `-n 1` - each task needs only 1 core to run `-W 10:00` - each task needs a maximum of 10 hours to run. This is in Days:Hours:Minutes `-J Pop_Sim[1-30]` - The job is going to be called `Pop_Sim` (this can be anything you want). The `[1-30]` means the job will be run 30 times, first with the index 1, then 2, and so on to 30. We will come back to this later. `-oo R-%J-%I.o` - The *log* file will be written to `R-<job number>-<index number (i.e. 1-30)>.o -eo R-%J-%I.o` - The *error* file will be written to `R-<job number>-<index number (i.e. 1-30)>.o R CMD BATCH "--args ${LSB_JOBINDEX}" JASMIN_parameters_file.R "console/console${LSB_JOBINDEX}.Rout"` - This dictates what is to be run when the job starts. This says, use R to run my R script `JASMIN_parameters_file.R`, sending the job index (1-30) to the script as an argument `--args ${LSB_JOBINDEX}`, and write all console output to `console/console${LSB_JOBINDEX}.Rout`.

This is quite a lot to take in so take a moment to familiarize yourself with what this files means

How would you change the file to fit these new requirements:

- Your job name is now 'My_analysis'
- Your wall time is 30 hours (can you still use the same queue?)
- Your memory requirement is 8GB

Changing the R script for JASMIN

We need to make one small change to our R script so that it will work on JASMIN (available at `JASMIN_parameters_file.R`). When we used a loop we set `i` to increase by one each time we passed though the loop so that we ran the model for each parameter set. Now we want our R script to use the value given to it by the scheduler, that's the `--args ${LSB_JOBINDEX}` bit of the job script.

The only change we need to make is to remove the loop and swap it out for this line:

```
i <- as.numeric(commandArgs(trailingOnly = TRUE))[1]
```

This line takes the index value from the job file (1, 2, ..., 30) and then proceeds as in the `for` loop to get the required row from the parameters file. We use the parameters file to parameterise our simulation however you may use it differently in your own work. For example you might instead use a list of species names (running a model for each) or a list of file names (to reformat or analyse).

Here is what our new script looks like:

```
source("scripts/simFunctions.R")
library(ggplot2)
library(plyr)

kNumReplications <- 5

# Size of the area
areaW <- 30
areaH <- 30

# Homesteaders, they don't care about finding a neighbor
# only used in random seeding
numPioneers <- 30

# Get job index
i <- as.numeric(commandArgs(trailingOnly = TRUE))[1]

parameters <- read.csv('data/parameters.csv')
```

```
# Record the time so we know how long it takes
start <- Sys.time()

cat('\n\nStarting parameter set', i, '\n')

#These are the people who follow the pioneers
kNumSettlers <- parameters[i, 'kNumSettlers']
# How many scouting attempts will settlers make, before abandoning
kMaxLookAroundSteps <- parameters[i, 'kMaxLookAroundSteps']

#Try out different seeding rules
# (1:Random) (2: Central Square) (3:Central Ring) ( 4: Two Columns)
seeding.opt <- parameters[i, 'seeding.opt']
# (1: Random) (2:NEWS only) (3:Diagonal settling only)
settling.option <- parameters[i, 'settling.option']

adjacells <- getAdjacentCellsDataFrame()

# Make multiple runs (Replication of simulation) and take the average of stats
st <- data.frame()
st_row <- vector()
for(i in 1:kNumReplications) {
  area_df <- resetIteration()

  seedAreaWithPioneers(numPioneers,seeding.opt)
  simstats <- accommodateSettlers(kNumSettlers, settling.option)

  found.home <- simstats[1]
  max.look.around <- simstats[2]
  #compute for this iterations
  st_row <- store_iteration_stats(i, kNumSettlers, found.home, max.look.around)
  st <- rbind(st,st_row)
}

# Render plot
plotname <- paste0('Pop_map_',
                   paste(kNumSettlers, kMaxLookAroundSteps, seeding.opt, settling.option, sep = '_'),
                   '.png')
png(filename = file.path('output_figures', plotname),
    width = 6, height = 4, units = 'in', res = 300)
  drawArea(area_df)
dev.off()
names(st) <- c("Iter", "FoundHome", "NumSettlers", "Percent")
tablename <- paste0('Pop_table_',
                    paste(kNumSettlers, kMaxLookAroundSteps, seeding.opt, settling.option, sep = '_'),
                   '.csv')
write.csv(st, file = file.path('output_tables', tablename), row.names = FALSE)
```

When I ran this across all 300 rows on JASMIN I had up to 80 cores at a time and it took 3 minutes to run.

- Can you identify the key new line we have added?
- If this now runs in 3 minutes was it worth it?

Installing R packages

We are almost ready to run our scripts. We have the instructions for our job in the `pop_sim.job`, and we have our R code in `JASMIN_parameter_file.R`. One common problem we need to overcome is the installation of R packages. The R packages on you PC may not be installed on JASMIN so you will need to install then first. All the packages you install are available to you alone, this means that the packages other people install don't effect you and vica versa.

You can install packages in the same way you would on you PC. First, at the Linux terminal type `R` and press return. This will start up R.

```
# Now we use R jus as we would normally
install.packages('ggplot2')
install.packages('plyr')
install.packages('reshape')
```

You might see that you get a lot more text to console than you would normally see on your windows PC. This is because installation on a Linux machine is a little different, nothing to worry about!

Submitting our job

Now that we have our R script ready and the job script written we can run our job. First, make sure your output folders are empty. Second, make sure you are in the directory with the `pop_sim.job` file. When here the job can be submitted with `bsub < pop_sim.job` at the Linux terminal. This says, submit my job (`pop_sim.job`) to the queue system (`bsub`)

```
bsub < pop_sim.job
```

If the job is submitted successfully you will then get a message like:

```
Job <XXXXXXXX> is submitted to queue <short-serial>
```

This long number is the unique identifier for you job and can be used to do various things. A useful one to know is, `bkill -r XXXXXXXXXX`, which will kill your job immediately (good to know if you have made a mistake).

You can monitor the progress of your job by typing `busers`. This command shows a summary of your activity, with columns for the total number of jobs you have on JASMIN (`njobs`), and how many of these are pending (`pend`), versus running (`run`).

If things go well then output will appear in the two output folders. If things go wrong and output does not appear or looks wrong we need to debug.

Hopefully that failed, so now we have an opportunity to go through debugging

Debugging errors

In our job file there were two lines that specified where log and error files would be written:

```
#BSUB -oo R-%J-%I.o
#BSUB -eo R-%J-%I.e
```

These are the first files we want to check to see what has gone wrong. These files will tell us if anything went wrong in the scheduling of the job.

Open up one of your `.e` files and you should find this error

```
/usr/lib64/R/bin/BATCH: line 60: console/console287.Rout: No such file or directory
```

This saying that the job script asked for console output to be written to the `console` directory but that directory does not exist so it stopped. We can fix this by creating a folder called `console`. once you have done this run your script again.

This time the scripts should run successfully. However, if there was an issue with the outputs we could look at the console outputs in the `console` folder. It is there that we would find reports of errors during the running of the R code.

Going that step further

If you have got this far it is time for you to start on your own projects. Hopefully you have brought something with you that you can work on. If you haven't, find someone who has and team up with them. If you don't want to work with anyone else here are some activities you could do with the example we have been working through. For the answers you will need to look at the JASMIN help documentation (https://help.jasmin.ac.uk/category/107-batch-computing-on-lotus).

- Copy all the output files back to your space on the CEH network
- How would you submit a job so that it only ran **after** another job has completed
- Change our log files so that they all write to the same file, without overwriting
- Run the job again so that it is limited to running only 2 tasks at the same time