Assignment 4

Junxiao Bu
999452701
SSH: git@bitbucket.org:shingtime/sta242-project2.git

May 14, 2015

PART I: Purpose and Organization

In this project, the C version of $runBMLGrid$ is implemented. The main purpose is comparing the running speeds' difference between R version and C version. Specificlly, the different grid size and density will be discussed separately. Besides, some verifications are present, which are related the consistency of R and C code.

Part II: Algorithm Introdcution

$CrunBMLGrid$, the C version of $runBMLGrid$, is implemented. The algorithm is same as what I implemented using R.In this simuation process, the blue cars will move upwards first. Then the red cars will move to the right. If cars move to the borders of the grid, red cars will jump back to the left border of the same row. Blue cars will jump back to the bottom border of the same column. All cars must move simultaneously. Here, the time step I defined in the package is a little bit different. In my definition, each time step means blue cars and red cars move once in their own orders.

To verify if the simualation results are same through $runBMLGrid$ and $CrunBMLGrid$. An 3*3 grid with 4 steps of movements are shown.
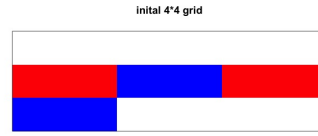The initial grid is shown first.

Figure 1: inital Grid of 4*4

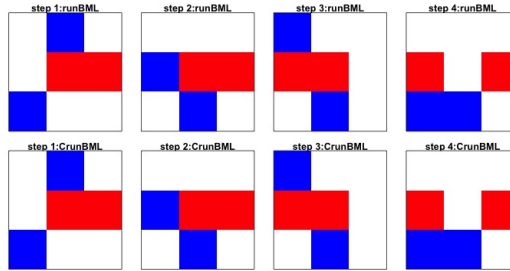All the plots that show the moving steps are shown below.

Figure 2: Movement via R and C

From the plots, one can observe the moving process by eyes and verify the result step by step. In this 4*4 grids, the moving results are same through $CrunBMLGrid$ or $runBMLGrid$.

For further confirmation, I simulated the process of 128*128 grid with density 0.5. The iteration is 100 times. The plots are shown below.
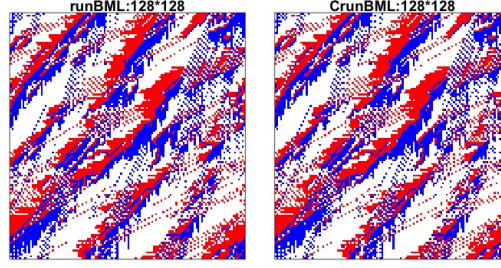
Figure 3: 128*128 grid via R and C

Part III: Prodfiling Code

As mentioned before, my R code is slower than the C code. The profiling tables of different are shown below. As the benchmark, I use a150*150 grid with 3000 number cars for each color. The iteration will be 100 times.

Table 1: Profiling of loop version in R

Time Type(seconds)	self.time	self.pct	total.time	total.pct
"movecars1"	63.76	85.45	74.58	99.95
"[[.data.frame"	2.08	2.79	6.44	8.63
"match"	1.38	1.85	2.08	2.79
"Anonymous"	1.14	1.53	1.36	1.82
"$"	0.88	1.18	8.76	11.74

Table 2: Profiling of vectorized version in R

Time Type(seconds)	self.time	self.pct	total.time	total.pct
"move_cars"	0.08	19.05	0.42	100.00
".change_color"	0.08	19.05	0.1	23.81
"which"	0.04	9.52	0.08	19.05
"cbind"	0.04	9.52	0.04	9.52
"car_coordinate"	0.02	4.76	0.2	47.62

The comparison of system time is shown below.

3

Table 3: Profiling of vectorized version in R

Time Type(seconds)	self.time	self.pct	total.time	total.pct
".C"	0.02	100	0.02	100

Table 4: Time for running different code

Time Type(seconds)	User Time	System Time	Elapsed Time
before vectorization	69.117	12.023	81.312
after vectorization	0.496	0.082	0.578
C version	0.010	0.001	0.011

From the above tables, by comparing the user time , the loop version of R code is definitely the slowest. The C code is fastest. There is a very significant speedup from the loop-version of the code.

So we get a speedup of 50 times by using C code. The C code significantly outperforms the vectorized version written in R. I This doesnt mean R is a bad language. One possible reason is that my vectorized version is not that efficient. The other one is that R's dynamic nature, which makes it easy to express computations, often leads to slower code than with compiled languages like C.

PART IV: Performance over Different Grid sizes

Next, the impacts of different grid size on the performance of the code are going to explored. The vectorized version R code and C code's performance will be discussed. First, the performance of the code over different grid sizes will be discussed. Here I use the grid sizes of 4*48*8, 16*16, 32*32, 64*64, 128*128, 256*256 and 512*512,and 1024*1024. And I choose three densities to check the difference. The first density is 0.2(traffic flow). The second density is 0.35(transition phase). The third density is 0.7(jam phase). The iteration will be 100 times. The plot is shown below.
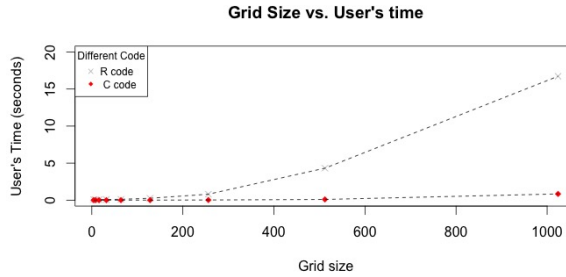

Figure 4: Density:0.2

For R code, these plot show that as the number of grid size, and the total number of cars, doubles, the time taken approximately doubles. But for C code, as grid sizes
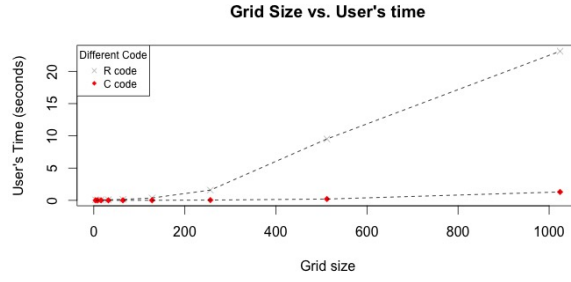
Grid Size vs. User's time

Figure 5: Density:0.35
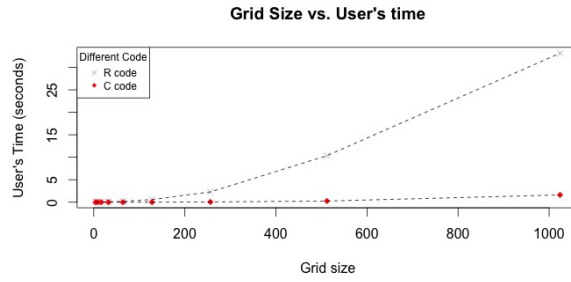
Grid Size vs. User's time

Figure 6: Density:0.7

increasing, the time taken approximately increases far less than grid sizes. The user time of C code are always significantly shorter than R code's. I calculate the ratio of different code's user time. It seems that the ratio is decreasing as the grid sizes increases. As grid sizes increases, this ratio may unchanged in a range. Due to the limitation of computing source, further computations are ignored.

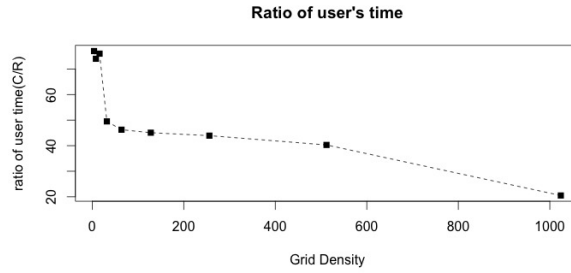**Ratio of user's time**

Figure 7: Density:0.7

PART V: Performance over Different Densities

Second, the performance of the code over different densities will be discussed. Here I use the grid size 64*64,128*128 and 256*256. The density is in the range of 0.1 to 0.7.(equally split in 13 blocks). The iteration will also be 100 times. The plots are shown below.
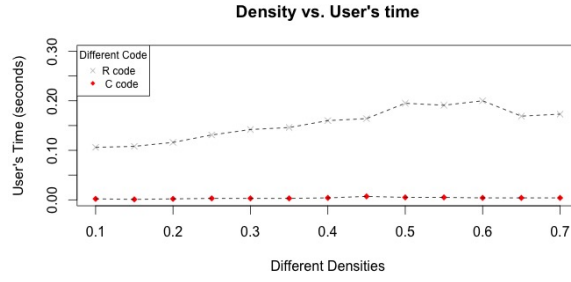
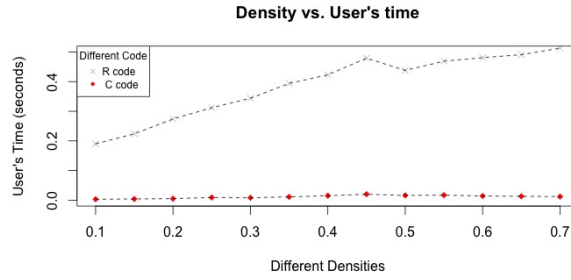Figure 8: 64*64 Grid with different densities

Figure 9: 128*128 Grid with different densities
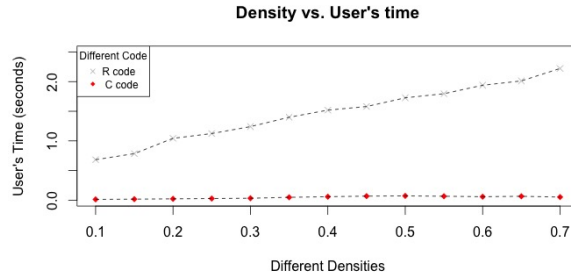
Figure 10: 256*256 Grid with different densities

For C code, these plot show that as the density increases, the time taken increases(not too significant). For R code, as densities increasing, the time taken approximately increases significantly. The performance of C code is always better than R code.

.

PART VI: Performance over Different iteration times

Next, the obvious relation between iteration time and user's time is explored. For both of C and R code, the user's time should increase as grid sizes increases. Here I choose 128*128 grid with density 0.35. The iteration times is in the range from 1000 to 10000 with interval 1000.

Obviously , the iteration time is linearly increasing with increasing in iteration

Iterations vs. User's time (R)
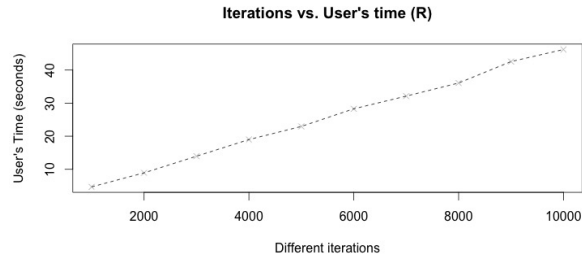
Figure 11: R's performance via different iteration times

Iterations vs. User's time (C)
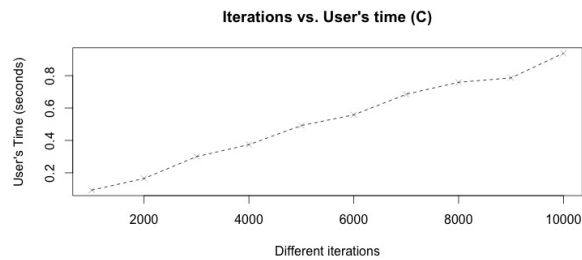
Figure 12: C's performance via different iteration times

times for both code. But C code's performance is far more efficient that R code's performance.

PART VII: Conclusion

This report explores the different performance of C code and R code implementing same algorithm. The first part verifies that the two sets of codes can generate same result.(Same grid). The next several parts discuss the performance between two methods. The possible reasons, about why R implementation is slowerr relatively to C version, are also discussed.In this project, I choose to use **.C()** to call c code. There are other options like **.call()** and **Rcpp**. Implementing these methods may be helpful for further optimize the performance. In this case, writing C code is worth. In real life project, it may be realistic to combine R and C to finish one task.

Reference

1. classnotes

2. Piazza: 268, 277,281.

Appendix

C code

```c
/* Appendix: C source code.
 * file: CrunBML.c
 * this file is the c version of moving cars simultaneously.
 */

#include <stdio.h>
#include "runBML.h"
#include <string.h>


/* private function PROTOTYPE*/
void move_cars(double *grid, int *dimension, int direction, int *
    location, int *number);


/* main routine*/
void crunBMLGrid (double *grid, int *dimension , int *red_loc, int *
    blue_loc,int *nred , int *nblue, int *numsteps)

{

    int i;
    /* store color */
    int dir;

    for (i = 0; i< *numsteps ; i++){
      /* move blue cars*/
        dir = 2;
        move_cars(grid, dimension,dir,blue_loc,nblue);
        /* move red cars*/
        dir = 1;
        move_cars(grid, dimension,dir,red_loc,nred);
    }
    return;
}


void move_cars(double *grid,int *dimension, int direction, int *
    location, int *number)
{
    int color = direction;
    int i,t,k;

    /* generate a array to store index of array represent of the
    original grid */
    int new_loc[2*number[0]];

    for (i=0; i < number[0]; i++){
        if (color == 2) { // move blue cars
        /* for blue cars, move upwards */
            new_loc[i]  = location[i]-1 ;
        /* column won't change */
            new_loc[i + number[0]] = location[i + number[0]];
         /* border: move from top to bottom */
            if (new_loc[i]<1) new_loc[i] = dimension[0];
        }

        else {  /* move red cars*/
            new_loc[i]  = location[i] ;
            /* move to the right */
            new_loc[i + number[0]] = location[i + number[0]]+1;
            /* border : move from left to right */
```

```
60              if ( new_loc[i + number[0]] > dimension[1]) new_loc[i +
       number[0]] = 1;
61           }
62
63      }
64
65      /* Check if a car can be moved, if it can be moved, change it's
       value to −1 indicates it can be moved. */
66
67      for(k=0; k < number[0];k++) {
68
69          /* Create an index to change location x row into grid index*/
70
71
72          int indice = location[k]−1+dimension[0]*(location[k+number
       [0]]−1);
73          int indice_new =new_loc[k]−1+dimension[0]*(new_loc[k+number
       [0]]−1) ;
74
75          /* Change the value of cars that can be moved into −1*/
76
77          if (grid[indice_new] == 0) grid[indice] = −1;
78      }
79
80      /* based on the index, move the cars */
81
82      for (t=0; t < number[0];t++) {
83          if ( grid[location[t]−1+dimension[0]*(location[t+number[0]]−1)
       ] == −1)
84          {
85            /* change the corresponding cars */
86              grid[new_loc[t]−1+dimension[0]*(new_loc[t+number[0]]−1)] =
        color;
87              grid[location[t]−1+dimension[0]*(location[t+number[0]]−1)]
       = 0;
88              location[t] = new_loc[t];
89              location[t+number[0]] = new_loc[t+number[0]] ;
90          }
91      }
92 }
```

R Soruce Code of Package

```
1
2 ################## STA 242 HW2 final version ##################
3 ### Appendix C: R source code
4
5 #####Note: In the source file, all the 0 mean white color on the grid.
        All the 1
6 #### represent red cars. All the 2 represent blue cars. These three
      constant are using
7 ### to represent color.
8
9 ########### function to create initial grid ############
10 createBMLGrid =function(r,c, ncars){
11   ## check if the cars' numbers are reasonable
12   if (sum(ncars)/(r*c) > 1) return ("There are too many cars.")
13   else{
14     ## generate the empty grid
15     grid = matrix(0, r, c)
16     pos = sample(1:length(grid), sum(ncars))
17     ## −1:blue;  1: red
18     grid[pos] = sample(rep(c(2, 1), ceiling(ncars)))[seq(along = pos)]
19   }
20   ## s3 method to define grid's class
21   class(grid) = c("BMLGrid", "matrix")
22   grid
```

```r
23  }
24
25  ########### function to store the cars' coordinates ###########
26  ## In this algorithms, all the cars will move simultaneouly
27  car_coordinate = function(matrix){
28    ## check cars' location
29    index = which(matrix!=0)
30    ## row number with cars
31    row_indice = row(matrix)[index]
32    ## column number with cars
33    col_indice = col(matrix)[index]
34    ## each car's position
35    position = t(rbind(row_indice, col_indice))
36    ## cars' coordinates on the initial grid
37    color = matrix[position]
38    data.frame(row_indice=row_indice,col_indice=col_indice,colors =
        color)
39  }
40
41  ########### function to move cars ####################
42  ### the arguments color specifies what color cars we want to move
43  .move_cars = function (matrix,color){
44    ## call car_coodinate function to extract cars coordinates
45    car_info = car_coordinate(matrix)
46    ## those color's index we want
47    indice = which(car_info$colors==color)
48    ## create a matrix to store specific color's cars' location
49    current_location = as.matrix(cbind(car_info$row_indice[indice],car_
        info$col_indice[indice]))
50    colnames(current_location) = c("row_indice","col_indice")
51    ## duplicate the current location matrix and use it to compute the
        next location
52    possible_next_location = current_location
53    if(color==1){
54      ## add one to all red cars' column indices— move to the right
55      possible_next_location[,"col_indice"] = possible_next_location[,"
        col_indice"] + 1
56      ## check if the red cars move to the right border of the grid, if
        yes, change row indices to one
57      possible_next_location[,"col_indice"][possible_next_location[,"col
        _indice"] > ncol(matrix)] = 1
58    }
59    ## blue cars:move upwards
60    else{
61      ## minus one to all blue cars' row indices— move upwards
62      possible_next_location[,"row_indice"] = possible_next_location[,"
        row_indice"] − 1
63      ## check if the blue cars move to the upper border of the grid, if
         yes, change row indices to the
64      ## number of columns of the initial grid.
65      possible_next_location[,"row_indice"][possible_next_location[,"row
        _indice"] < 1] = nrow(matrix)
66    }
67    ## call change_color function to move the cars
68    .change_color(current_location,possible_next_location,color,matrix)
69  }
70
71  .change_color = function (current_location,possible_next_location,
      color,matrix){
72    ## use all the new indices for the cars to subset the original grid
73    new_indice = matrix[possible_next_location]
74    ## check if the new positions are blank. If yes, change those blanks
         to the corresponding colors
75    matrix[possible_next_location[which(new_indice == 0),,drop=FALSE ]]
        = color
76    ## change those cars' current locations' colors to blank
77    matrix[current_location[which(new_indice == 0),,drop=FALSE ]] = 0
78    matrix
```

10

```r
79  }
80
81  ########## function to get the final grid
82  ## need to export all global functions and global variables
83  rumBMLGrid= function(numSteps,grid){
84    # Register cluster
85    for(i in 1:numSteps){
86      ##move blue car
87      grid = .move_cars(grid,2)
88      ## move red car
89      grid = .move_cars(grid,1)
90    }
91    grid
92  }
93
94  ######### C vrsion to move cars and get the final grid ##############
95
96  CrunBMLGrid = function(grid,numsteps){
97    location = car_coordinate(grid)
98    ### red cars indices
99    red = as.matrix(location[which(location$colors==1),1:2])
100   ### blue cars indices
101   blue = as.matrix(location[which(location$colors==2),1:2])
102   result = .C("crunBMLGrid",grid,dim(grid),red,blue,nrow(red),
103               nrow(blue),as.integer(numsteps))
104   ### return the final grid
105   result[[1]]
106
107 }
108 ######## plot function: S3 method ###########
109 plot.BMLGrid = function(x,...){
110   image_grid = matrix(match(x, c(0, 1, 2)),
111                       nrow(x), ncol(x))
112   ## the row number should be opposite
113   image_grid = image_grid[c(nrow(image_grid):1),]
114   ## get the correct order
115   image(t(image_grid), col = c("white", "red", "blue"),axes=FALSE,...)
116   box()
117
118 }
119
120 ###### summary function : S3 method ###########
121 #### this function exports the given color's grid's some summary
        statistics.
122 #### 1. how many cars for this color
123 #### 1. how many cars can move
124 #### 2. how many cars are blocked
125 #### 3. average velocity
126
127 summary.BMLGrid = function(object,...){
128   car_info_blue = car_coordinate(object)
129   output_blue = .blue_summary(object,car_info_blue)
130   ### red cars
131   grid_red = .move_cars(object,2)
132   car_info_red = car_coordinate(grid_red)
133   output_red = .red_summary(grid_red,car_info_red)
134   output = c(output_blue,output_red)
135   names(output) = c("total-blue","move-blue","block-blue","velocity-
        blue","total-red","move-red","block-red","velocity-red")
136   options(digits=4)
137   output
138 }
139 ######### function to summary blue car
140 .blue_summary = function(grid,car_info){
141   indice = which(car_info$colors == 2)
142   num_car = length(indice)
143   current_row = car_info$row_indice[indice]
144   current_col = car_info$col_indice[indice]
```

11

```r
145    next_col = current_col
146    next_row = current_row − 1
147    next_row[ next_row < 1 ]  = nrow(grid)
148    new_possible_position = cbind(next_row, next_col)
149    move = sum(grid[new_possible_position] == 0)
150    block = num_car − move
151    velocity = move/num_car
152    output = c(num_car, move, block, velocity)
153    output
154  }
155  ######### function to summary red car
156  .red_summary = function(grid, car_info){
157    indice = which(car_info$colors == 1)
158    num_car = length(indice)
159    current_row = car_info$row_indice[indice]
160    current_col = car_info$col_indice[indice]
161    next_row = current_row
162    next_col = current_col + 1
163    next_col[ next_col > ncol(grid) ]  = 1
164    new_possible_position = cbind(next_row, next_col)
165    move = sum(grid[new_possible_position] == 0)
166    block = num_car − move
167    velocity = move/num_car
168    output = c(num_car, move, block, velocity)
169    output
170  }
171  ########### function to plot a given series of densities' grids plot
172
173  .reproduce_plot = function(r, c, density, numSteps){
174    car_num = sapply(density, function(x) round(r*c*x/2))
175    initial = lapply(array(car_num), function(x)createBMLGrid(r, c, c(red
         = car_num[[x]], blue = car_num[[x]])))
176    final = lapply(1:length(density), function(x) rumBMLGrid(numSteps,
         initial[[x]]))
177    par(mfrow = c(2, 3), mar = c(1,1,1,1), pty = "s")
178    sapply(1:length(density), function(x) plot(final[[x]], main = paste(
         "density = ", density[x])))
179  }
180
181
182  ######### this function is to get the average velocity over different
         density ###########
183  velocity_density = function(r,c,density,numSteps){
184    car_num = sapply(density, function(x) round(r*c*x/2))
185    initial = lapply(1:length(density),function(x)createBMLGrid(r, c,
186                                                   c(red =
         car_num[[x]], blue = car_num[[x]])))
187    final = lapply(1:length(density),function(x){
188      rumBMLGrid(numSteps,initial[[x]])
189    })
190    veo_summary = lapply(1:length(density),function(x){
191      summary(final[[x]])
192    })
193    ## list to store red cars and blue cars' average velocity across
         different density
194    veo_summary = do.call(rbind.data.frame,veo_summary)
195    velocity = cbind((veo_summary[,2]+veo_summary[,6])/(veo_summary[,1]+
         veo_summary[,5]),density)
196    colnames(velocity) = c("velocity","density")
197    as.data.frame(velocity)
198  }
199
200
201  ######### SLOWER VERSION OF move_cars & rumBMLGrid
         ###############################
202
203  .move_cars1 = function(matrix){
204    car_info = car_coordinate(matrix)
```

```r
205    red_indice = which(car_info$colors==1)
206    blue_indice = which(car_info$colors==2)
207    ## red cars: move to the right
208    ## check if we can move the red cars
209    for (i in red_indice){
210      current_row = car_info$row_indice[i]
211      current_col = car_info$col_indice[i]
212      next_row = current_row
213      if(current_col==ncol(matrix)) next_col = 1
214      else next_col = current_col +1
215      ## change the position
216      if(matrix[next_row,next_col] ==0){
217        matrix[next_row,next_col]=1
218        matrix[current_row,current_col]=0
219      }
220    }
221    ## blue cars:move upwards
222    for (j in blue_indice){
223      current_row = car_info$row_indice[j]
224      current_col = car_info$col_indice[j]
225      next_col = current_col
226      if(current_row==1) next_row = nrow(matrix)
227      else next_row = current_row - 1
228      ## change the position
229      if(matrix[next_row,next_col]==0){
230        matrix[next_row,next_col]=2
231        matrix[current_row,current_col]=0
232      }
233    }
234    matrix
235 }
236
237 .rumBMLGrid1= function(initial_grid,numSteps){
238    for (i in 1:numSteps){
239      update_grid = .move_cars1(initial_grid)
240      update_grid = .move_cars1(update_grid)
241      initial_grid = update_grid
242    }
243    initial_grid
244 }
```

R Soruce Code to Generate Figures and Tables in the Report

```r
1
2
3
4
5  ### Appendix B
6  ########## source code to reproduce the plot in the report.
7
8
9  #### figure1: 4*4 grid
10 dev.off()
11 par(mfrow=c(1,1))
12 g = createBMLGrid(3,3,c(2,2))
13 plot(g,main="inital 4*4 grid")
14 par(mfrow=c(2,4),mar=c(1,1,1,1))
15
16 #### figure 2: moving 4 steps
17 dev.off()
18 par(mfrow=c(2,4),mar=c(1,1,1,1))
19 plot(rumBMLGrid(1,g),main="step 1:runBML")
20 plot(rumBMLGrid(2,g),main="step 2:runBML")
21 plot(rumBMLGrid(3,g),main="step 3:runBML")
22 plot(rumBMLGrid(4,g),main="step 4:runBML")
23
24
```

```r
25  plot(CrunBMLGrid(g,1),main="step 1:CrunBML")
26  plot(CrunBMLGrid(g,2),main="step 2:CrunBML")
27  plot(CrunBMLGrid(g,3),main="step 3:CrunBML")
28  plot(CrunBMLGrid(g,4),main="step 4:CrunBML")
29
30  ####figure 3: 128*128 density=0.5 timestep=100
31  dev.off()
32  par(mfrow=c(1,2),mar=c(1,1,1,1))
33  g = createBMLGrid(128,128,c(128*128*0.5/2,128*128*0.5/2))
34  plot(rumBMLGrid(100,g),main="runBML:128*128")
35  plot(CrunBMLGrid(g,100),main="CrunBML:128*128")
36
37  ##### table r profiling
38
39  Rprof("/tmp/r_BML.prof")
40  g = createBMLGrid(150,150,c(3000,3000))
41  ss = rumBMLGrid(100,g)
42  Rprof(NULL)
43  head(summaryRprof("/tmp/r_BML.prof")$by.self, 5)
44
45  Rprof("/tmp/C_BML.prof")
46  g = createBMLGrid(150,150,c(3000,3000))
47  ss = CrunBMLGrid(g,100)
48  Rprof(NULL)
49  head(summaryRprof("/tmp/C_BML.prof")$by.self, 10)
50  system.time(rumBMLGrid(100,g))
51  system.time(CrunBMLGrid(g,100))
52
53
54  ###### figure 4:
55
56
57
58  ######## performance with different grid size
59
60
61  ### grid size with user's time
62  #### suppose the density is 0.2 ,0.35 , 0.7
63  #### run 100 hundered times
64
65  N = 2^(2:10)
66  timings =sapply(N,
67          function(n) {
68              print(n)
69              g = createBMLGrid(n,n,c(n^2*0.2/2,n^2*0.2/2))
70              system.time(rumBMLGrid(100,g))
71          })
72
73  timings_C =
74    sapply(N,
75          function(n) {
76              print(n)
77              g = createBMLGrid(n,n,c(n^2*0.2/2,n^2*0.2/2))
78              system.time(CrunBMLGrid(g,100))
79          })
80
81  dev.off()
82  plot(N, timings[1,], type = "p",
83        xlab = "Grid size" , ylim = c(0,20) ,
84        ylab = "User's Time (seconds)",pch=4,col=24,main="Grid Size vs.
      User's time")
85  lines(N,timings[1,],lty=2)
86  points(N,timings_C[1,],pch=18,col=58)
87  lines(N,timings_C[1,],lty=20)
88
89  legend("topleft",title="Different Code",c("R code"," C code"),
90          pch=c(4,18),col=c(24,58),cex=0.75)
91
```

14

```r
timings_0.35 = sapply(N,
                      function(n) {
                        print(n)
                        g = createBMLGrid(n,n,c(n^2*0.35/2,n^2*0.35/2)
                        )
                        system.time(rumBMLGrid(100,g))
                      })

timings_C_0.35 =
  sapply(N,
         function(n) {
           print(n)
           g = createBMLGrid(n,n,c(n^2*0.35/2,n^2*0.35/2))
           system.time(CrunBMLGrid(g,100))
         })


plot(N, timings_0.35[1,], type = "p",
     xlab = "Grid size",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Grid Size vs.
     User's time")
lines(N,timings_0.35[1,],lty=2)
points(N,timings_C_0.35[1,],pch=18,col=58)
lines(N,timings_C_0.35[1,],lty=20)
legend("topleft",title="Different Code",c("R code","C code"),
       pch=c(4,18),col=c(24,58),cex=0.75)



timings_0.7 = sapply(N,
                     function(n) {
                       print(n)
                       g = createBMLGrid(n,n,c(n^2*0.7/2,n^2*0.7/2))
                       system.time(rumBMLGrid(100,g))
                     })

timings_C_0.7 =
  sapply(N,
         function(n) {
           print(n)
           g = createBMLGrid(n,n,c(n^2*0.7/2,n^2*0.7/2))
           system.time(CrunBMLGrid(g,100))
         })

plot(N, timings_0.7[1,], type = "p",
     xlab = "Grid size",
     ylab = "User's Time (seconds)",pch=4,col=24,main="Grid Size vs.
     User's time")
lines(N,timings_0.7[1,],lty=2)
points(N,timings_C_0.7[1,],pch=18,col=58)
lines(N,timings_C_0.7[1,],lty=20)
legend("topleft",title="Different Code",c("R code","C code"),
       pch=c(4,18),col=c(24,58),cex=0.75)

ratio = timings_0.7[1,]/timings_C_0.7[1,]


plot(N,ratio,xlab = "Grid Density",ylab = "ratio of user time(C/R)",,
     type="p",pch=15,
     main = "Ratio of user's time")
lines(N,ratio,lty=2)

######## different density

density = seq(from=0.1,to=0.7,length.out = 13)
```

```r
156 timing_density_128 =
157    sapply(density,
158           function(n) {
159              print(n)
160              g = createBMLGrid(128,128,c(red = ceiling(128^2*n/2),blue =
       ceiling(128^2*n/2)))
161              system.time(rumBMLGrid(100,g))
162           })
163
164
165 timings_density_C_128 =
166    sapply(density,
167           function(n) {
168              print(n)
169              g = createBMLGrid(128,128,c(red = ceiling(128^2*n/2),blue =
       ceiling(128^2*n/2)))
170              system.time(CrunBMLGrid(g,100))
171           })
172
173
174 plot(density, timing_density_128[1,], type = "p",
175        xlab = "Different Densities",
176        ylab = "User's Time (seconds)",pch=4,col=24,main="Density vs.
       User's time",ylim = c(0,0.5))
177 lines(density,timing_density_128[1,],lty=2)
178 points(density,timings_density_C_128[1,],pch=18,col=58)
179 lines(density,timings_density_C_128[1,],lty=20)
180
181 legend("topleft",title="Different Code",c("R code"," C code"),
182          pch=c(4,18),col=c(24,58),cex=0.75)
183
184 ##### 64*64 grid size
185
186 timing_density_64 =
187    sapply(density,
188           function(n) {
189              print(n)
190              g = createBMLGrid(64,64,c(red = ceiling(64^2*n/2),blue =
       ceiling(64^2*n/2)))
191              system.time(rumBMLGrid(100,g))
192           })
193
194 timings_density_C_64 =
195    sapply(density,
196           function(n) {
197              print(n)
198              g = createBMLGrid(64,64,c(red = ceiling(64^2*n/2),blue =
       ceiling(64^2*n/2)))
199              system.time(CrunBMLGrid(g,100))
200           })
201
202
203 plot(density, timing_density_64[1,], type = "p",
204        xlab = "Different Densities",
205        ylab = "User's Time (seconds)",pch=4,col=24,main="Density vs.
       User's time",ylim = c(0,0.3))
206 lines(density,timing_density_64[1,],lty=2)
207 points(density,timings_density_C_64[1,],pch=18,col=58)
208 lines(density,timings_density_C_64[1,],lty=20)
209 legend("topleft",title="Different Code",c("R code"," C code"),
210          pch=c(4,18),col=c(24,58),cex=0.75)
211
212 ##### 256*256 grid size
213
214 timing_density_256 =
215    sapply(density,
216           function(n) {
217              print(n)
```

```r
218              g = createBMLGrid(256,256,c(red = ceiling(256^2*n/2),blue =
          ceiling(256^2*n/2)))
219              system.time(rumBMLGrid(100,g))
220            })
221
222 timings_density_C_256 =
223    sapply(density,
224            function(n) {
225              print(n)
226              g = createBMLGrid(256,256,c(red = ceiling(256^2*n/2),blue =
          ceiling(256^2*n/2)))
227              system.time(CrunBMLGrid(g,100))
228            })
229
230 plot(density, timing_density_256[1,], type = "p",
231        xlab = "Different Densities",
232        ylab = "User's Time (seconds)",pch=4,col=24,main="Density vs.
          User's time",ylim = c(0,2.5))
233 lines(density,timing_density_256[1,],lty=2)
234 points(density,timings_density_C_256[1,],pch=18,col=58)
235 lines(density,timings_density_C_256[1,],lty=20)
236 legend("topleft",title="Different Code",c("R code"," C code"),
237          pch=c(4,18),col=c(24,58),cex=0.75)
238
239
240 ############## different iteration times
241
242 iteration = seq(1000,10000,1000)
243
244 timings_iteration =
245    sapply(iteration,
246            function(n) {
247              print(n)
248              g = createBMLGrid(128,128,c(red = ceiling(128^2*0.35/2),
          blue = ceiling(128^2*0.35/2)))
249              system.time(rumBMLGrid(n,g))
250            })
251
252
253 timings_iteration_C =
254    sapply(iteration,
255            function(n) {
256              print(n)
257              g = createBMLGrid(128,128,c(red = ceiling(128^2*0.35/2),
          blue = ceiling(128^2*0.35/2)))
258              system.time(CrunBMLGrid(g,n))
259            })
260
261 plot(iteration, timings_iteration[1,], type = "p",
262        xlab = "Different iterations",
263        ylab = "User's Time (seconds)",pch=4,col=24,main="Iterations vs.
          User's time (R)")
264 lines(iteration,timings_iteration[1,],lty=2)
265
266 plot(iteration, timings_iteration_C[1,], type = "p",
267        xlab = "Different iterations",
268        ylab = "User's Time (seconds)",pch=4,col=24,main="Iterations vs.
          User's time (C)")
269 lines(iteration,timings_iteration_C[1,],lty=2)
```

17