Biham-Middleton-Levine traffic model[1]

Interfacing C language in R

University of California, Davis
Qiwei Li

Background

The Biham–Middleton–Levine (BML) traffic model is a self-organizing structure. It consists some blue cars, some red cars, and a grid that cars move on.

At t = 0, all cars are randomly put on the grid. In our configuration, the blue cars move at time periods t = 1, 3, 5, ... and the red cars move at time periods t = 2, 4, 6, ..., i.e., they alternate in time. The “blue” cars move vertically upward. The “red” cars move horizontally rightwards. When a car reaches to the edge, it “warps” around in the same row or column. A car cannot move to cell if that cells is already occupied by another car (of any color).

Motivation

In my previous report[2] , I discussed different implementations of BML simulation with R. I demonstrate the significant advantage of vectorize functionality of R over the traditional for loop. However, when the grid size and number of steps of the simulation is large enough, the computation still takes a large amount of time. Fortunately, R provides the feature to interface with another programming language such as the C language, which is very low level but fast. This report is to explore the C implementation.

Implementation

R and C exhibit different computation models. R is a functional programming language. The main characteristic is that new values are always returned though functions in R. The advantage is that this helps to keep existing variable invariant. The disadvantage is R copies each variable needed in each function call and assign the return value each time when the function finishes. This can take significant amount of time.

On the other hand, C language allows to directly change the value at a particular memory location. This is very low level but powerful and efficient. In case of BML simulation, the grid will be represented as a vector. When cars move, values that represent grid positions are changed through pointers directly. This is very efficient. C implementation is expected to be much faster than R implementation.

---

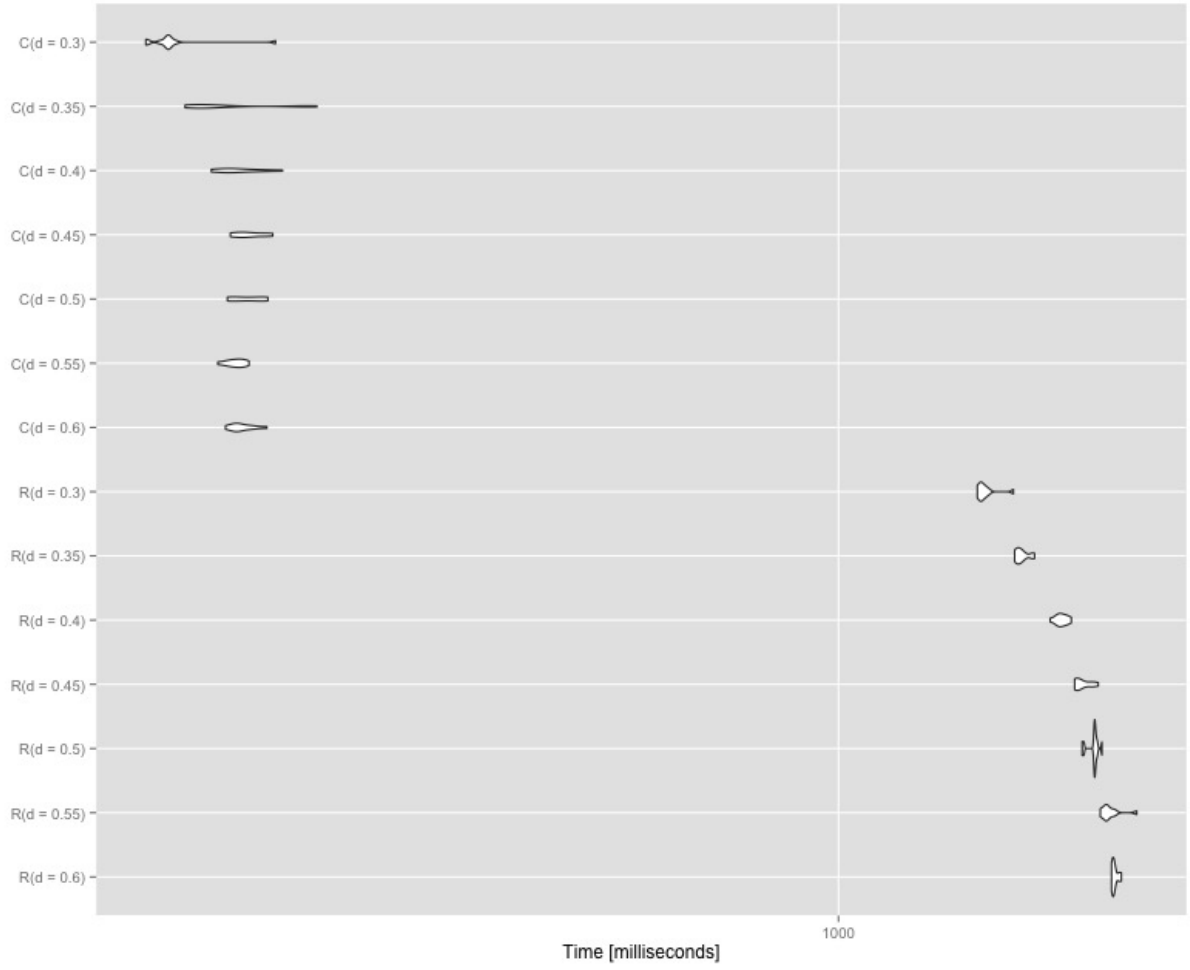
[1] Bitbucket repository: `https://qiwei_li@bitbucket.org/qiwei_li/sta242-assignment4`
[2] Previous report is available at `https://github.com/qiwei-li/biham-middleton-levine-traffic-model`

Performance

In our previous report, we examined that the R implementation would take longer time when the grid size or density increases. Now, let's take a look if C implementation[3] exhibits similar behavior.

Density

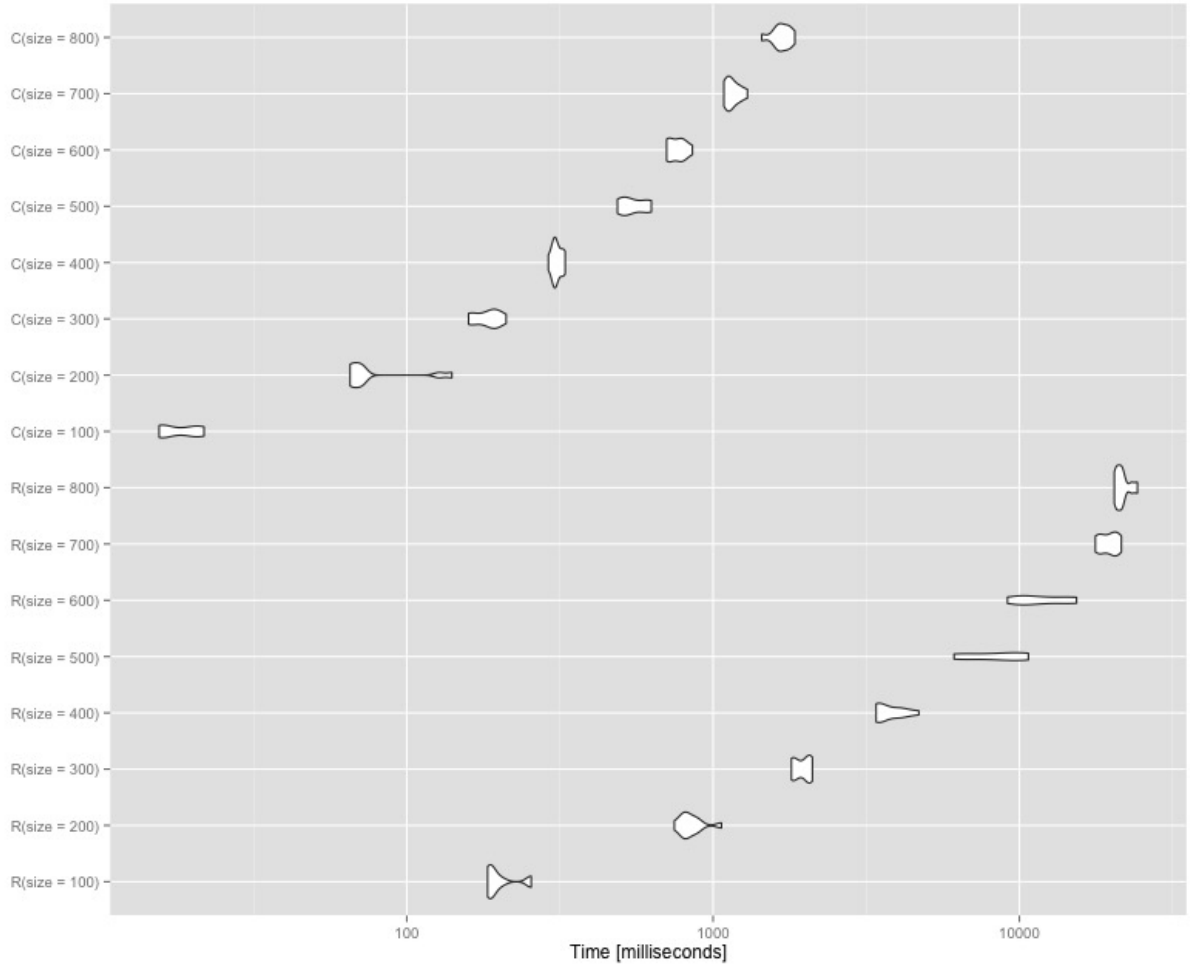
As expected, C implementation is much faster. For different density, the computation time has similar trend when density increases. For densities that outputs a "free flow" stage for the BML movements, time taken increases when density increases. For densities that outputs a "global jam" stage for the BML movements, time taken reaches to a certain level and stops.



[3] See Appendix I for code

Grid size

Again, C implementation is much faster. In the plot below, we tried grid size of 100*100, 200*200, 300*300, … For different grid size, but computation time has similar trend when grid size increases. While grid sizes increase in squares, the computation time increases in squares. However, one should notice that time values for R implementation is flatter. This indicates that R implementation has a larger overhead.



Limitations

One might ask, if C is always more efficient, why not implement everything in C?

If we are only considering the computation time, the answer is we should implement everything in C. However, for some tasks, R is much easier to implement and the value of return (computation advantage) of C may not be worth it anymore.

For example, I do not want to implement **createBMLGrid()** function in C. In R, a simple **sample()** function will complete most of the task. In C, one has to keep generating values

randomly and keep track how many values are generated. I only use createBMLGrid() once in a while and it is so much easier to implement in R. Why waste so much time implement it in C?

The conclusion is if a task is needed repeatedly and it involves lots of iterations and calculations, implement in the low level language C will save a lot computation time and will be worth it. Otherwise, use the high-level language R!

Appendix I - code

```
crunBMLGrid = function(grid, numSteps){
  r = nrow(grid)
  c = ncol(grid)
  a = .C("crun1", as.integer(grid), as.integer(r), as.integer(c), as.integer(numSteps))
  matrix(a[[1]], nrow=nrow(grid))
}

void cmoveBlue1(int *mat, int*nRows, int*nCols)
{
  int total = (*nRows)*(*nCols);
  int omat[total];
  for(int i=0; i<total; i++)
  {
    omat[i]=mat[i];
  }

  for(int r=0; r<(*nRows); r++)
  {
    for(int c=0; c<(*nCols); c++)
    {
      if(omat[c*(*nRows)+r]==2)
      {
        if(r==0)
        {
          if(omat[(c+1)*(*nRows)-1]==0)
          {
            mat[(c+1)*(*nRows)-1]=2;
            mat[c*(*nRows)+r]=0;
          }
        }
        else
        {
          if(omat[c*(*nRows)+r-1]==0)
          {
            mat[c*(*nRows)+r-1]=2;
            mat[c*(*nRows)+r]=0;
          }
        }
      }
    }
  }
}
```

```
void cmoveRed1(int *mat, int *nRows, int *nCols)
{
  int total = (*nRows) * (*nCols);
  int omat[total];
  for(int i=0; i<total; i++)
  {
    omat[i]=mat[i];
  }
  for(int r=0; r<(*nRows); r++)
  {
    for(int c=0; c<(*nCols); c++)
    {
      if(omat[c*(*nRows)+r]==1)
      {
        if((c+1)%(*nCols)==0)
        {
          if(omat[r]==0)
          {
            mat[r]=1;
            mat[c*(*nRows)+r]=0;
          }
        }
        else
        {
          if(omat[(c+1)*(*nRows)+r]==0)
          {
            mat[(c+1)*(*nRows)+r]=1;
            mat[c*(*nRows)+r]=0;
          }
        }
      }
    }
  }
}

void crun1(int *mat, int *nRows, int *nCols, int *T)
{
  if((*T)%2==0)
  {
    for(int i=0; i<((*T)/2); i++)
    {
      cmoveBlue1(mat, nRows, nCols);
      cmoveRed1(mat, nRows, nCols);
    }
  }
  else
  {
    for(int i=0; i<((*T)/2); i++)
    {
      cmoveBlue1(mat, nRows, nCols);
      cmoveRed1(mat, nRows, nCols);
    }
    cmoveBlue1(mat, nRows, nCols);
  }
}
```

Appendix II - documentation and test functions

```
\name{crunBMLGrid}
\alias{crunBMLGrid}
\title{crunBMLGrid}

\description{
Simulates BML model interfacing C code
}

\usage{
crunBMLGrid(grid, numSteps)
}

\arguments{
\item{grid}{A BML grid}
\item{numSteps}{An integer representing number of moves of the simulation}
}

\value{
return the final grid
}

\examples{
grid = createBMLGrid(r = 67, c = 127,  density = 0.5)
g = crunBMLGrid(grid, numSteps = 100)
}

\keyword{BML}

grid = createBMLGrid(r = 100, c = 100, density = 0.5)
a = runBMLGrid(grid = grid,numSteps = 100,ifPlot = FALSE,method = "fast")
b = crunBMLGrid(grid=grid, numSteps = 100)

if(sum(a!=b)!=0)
  stop("c routine is wrong")
```