# BML Model with C Code

**Hong Fan 912524085**

## 1. Introduction and motivation

In the assignment 2, Biham-Midleton-Levine traffic model is simulated in R and we investigated traffic behavior through adjusting car's densities and grid sizes. Through profiling process, bottlenecks are pined down and we refined our code step by step. Through vectorization, total running time spent on the simulation process greatly decreased by applying vectorization.

In the BML traffic model, there are two main functions: CreateBMLgrid and RunBMLgrid. RunBMLgrid is a time consuming function especially when the grid size and number of steps are very large. Due to predefined rules in this model that cars cannot advance when their target positions are occupied, we cannot simply apply vectorization. Thus my final version of RunBMLgrid still contains for-loops over the parameter NumSteps.

The following table shows profiling result of function "RunBMLgrid" from one simulation. In this simulation, the number of red cars and the number of blue cars are selected to be equal: 30 cars for each color.

| Function & components | Total.time(sec) | Total.pct | Self.time(sec) | Self.pct |
|---|---|---|---|---|
| RunBMLgrid | 24.76 | 100.00 | 0.12 | 0.48 |
| MoveCars | 24.64 | 99.52 | 1.40 | 5.65 |
| Location_car | 20.60 | 83.20 | 13.42 | 54.20 |

**Table 1 Partial Results of Profiling with Grid Size 100 by 100 (30 cars for each color and NumSteps is set to be 10000)**

As it is turned out, more than 99 percent of the time is spent on MoveCars. Part of the reason is that this simulation contains 10000 loops and during each loop the function needs to run MoveCars once. The result above also suggests focusing on MoveCars when we try to further refine the code of RunBMLgrid.

## 2. Implementing the BML Model in C

After profiling the code and vectorized operations, it is suggested to implement different computations using a compiled language C. In this section, we focus on implementing the bottleneck function MoveCars() and RunBMLgrid() as a C rountine.

### 2.1 Basic Ideas Behind MoveCars() in R and C

In my R code of MoveCars(), the basic idea is as follows:

1. Get locations of each car in the grid matrix.
2. Calculate its next locations based on the predefined rules for different colors.
3. Update its locations if its target position is not occupied (1 is for empty space, 2 for red cars and 3 for blue cars.).

In my C code of MoveCars(), the basic idea is very similar and this function contains 3 loops.

1. During each step, there are 2 small loops (alternatively) to move cars of different colors based on predefined rules.
   In each small for-loop (loop over same color cars) :
   1) First get locations of the car to be moved;
   2) Based on predefined rules to move the car.
   3) Update the "grid" (not in matrix form in C) at car level

   After looping over all the cars which have same colors, the old grid is updated at color level (the old grid is updated after each time step).
2. Loop over NumSteps.

Since C code MoveCars() contains the for-loop over NumSteps and the corresponding for-loop in R is in the RunBMLgrid(), we make comparison between RunBMLgrid() and crunBMLGrid(). Although cars with same colors move simultaneously in MoveCars() in R while same color cars move one by one in the c code function, it is fair to make comparison between them because the behind-the-scenes code of vectorized operations in R is approximately a for loop in C.

## 2.2 Performance Exploration by Using C Code

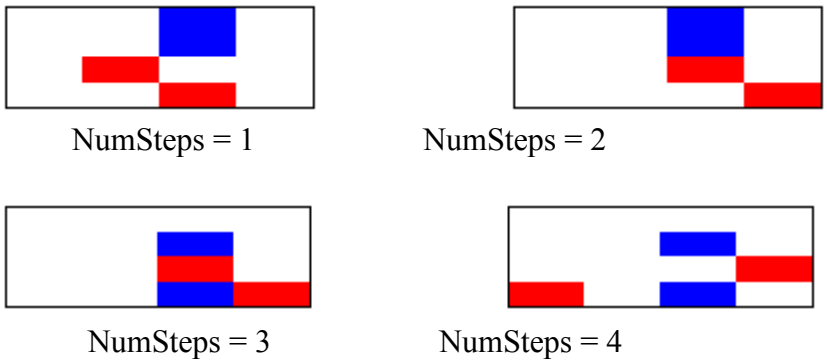


**Figure 1 Comparisons between Continuous Steps (4 by 4 grid)**

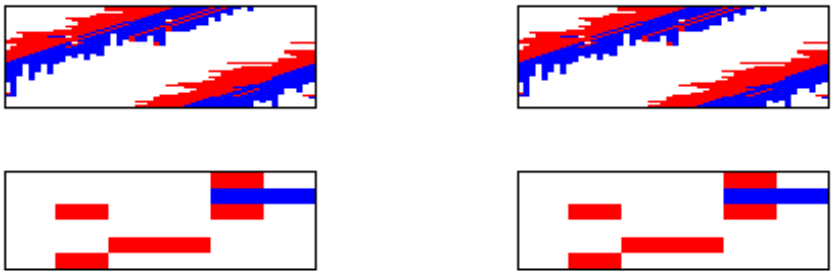

**Figure 2 Comparisons between RunBMLgrid() (left) and crunBMLGrid() (right)**
Top: grid size 50 by 50, blue cars = red cars = 500
Bottom: grid size 6 by 6, blue cars = 2, red cars = 6

In order to check the correctness of my crunBMLGrid(), I plotted 2 figures. Figure 1 shows the movement of cars on a 4 by 4 grid from NumSteps = 1 to 4 (simulated by crunBMLGrid(). It is clear to see that cars' movements obey the rules: when NumSteps is odd, blue cars move upward;

they will advance to the bottom when they reach the top grid but stay in the same columns. When NumSteps is even, red cars move rightward; they will advance to the beginning of their rows when they reach to the right edge of the grid but stay in the same rows.

Figure 2 shows the comparison between the simulation results from runBMLGrid() and crunBMLGrid(). It is clear to see that when grid size, the number of cars for each color and steps are specified, simulation results are the same.

Figure 3 shows comparison results regarding elapsed time by using runBMLGrid() and crunBMLGrid(). We fixed parameter NumSteps = 10000. In the left figure, total time spent on runBMLGrid() is increasing as density increases and has roughly linear relationship with density of cars. Although total elapsed time spent on runBMLGrid() at density =0.8 is almost 3 times as large as that when density = 0.2, total elapsed time spent on crunBMLGrid() over different densities keeps stable and is just slightly above 0.

When density is fixed at 0.8 and NumSteps is set to be 10000, total time spent on runBMLGrid() also shows strong positive relations with grid sizes, while total elapsed time spent on crunBMLGrid() over different grid sizes keeps stable and is slightly above 0. Figure 3 strongly supports the performance of C code and it is clear to see that at different densities and grid sizes, C code of the function crunBMLGrid() is much more efficient than runBMLGrid().

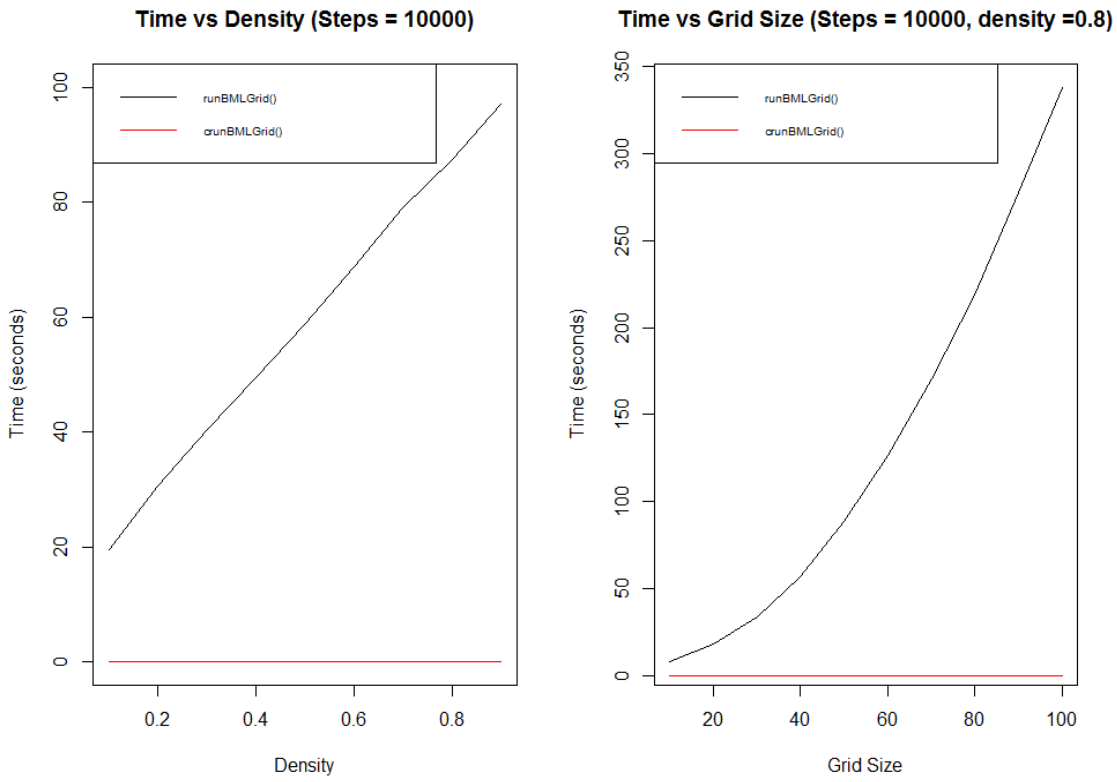


**Figure 3 Comparisons between runBMLGrid() and crunBMLGrid().**
**Left: Total time vs Density**
**Right: Total time vs Grid size**