

A Parallel Simulation of Traffic

Cynthia Bailey Lee

Prof. Scott Baden

December 6, 2002

Abstract

A traffic congestion simulator of particles in a 2-dimensional plane is presented. The simulation consists of particles (cars) that move freely in space according to their velocity. They are initialized with a distribution of velocities. When an area of the plane becomes congested, the cars in that area slow down, aggravating the congestion. This corresponds to a traffic jam: when too many cars are in one space they move more slowly. Congested areas can lead to load imbalances across the processors performing the simulation. The nature of these imbalances is analyzed, and a simple load-balancing scheme for this problem is presented. Finally, possible future improvements to the simulator and load-balancer designs are suggested.

1. Introduction

In parallel programming, this problem belongs to the class of ‘particle’ problems. In contrast to a problem like an Ordinary Differential Equation (ODE) solver, where a computation is done on each element of a two-dimensional matrix [1], these problems have a sparse representation that consists of a list of all the particles, each storing its own information such as its current location and velocity. Thus the space exists only implicitly as the boundaries on the space containing the particles. While matrices are naturally stored in an array, the particles may be stored structures such as lists or hashes.

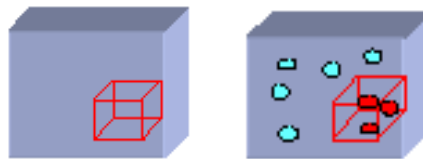


Figure 1: Contrast of 3-Dimensional Array and Particle Problems

Figure 1 [2] shows the spaces for a sample 3-dimensional array, and that of a particle problem. Notice that the distribution of elements is completely uniform in the array, but the particles may not uniformly distributed. The uneven distribution of particles in space can give rise to load imbalances across processors when the problem is decomposed by region of the space (such as the red cubes in the drawing). This adds to the difficulty of efficiently parallelizing these problems.

2. The Simulation

2.1 Specification

The simulation consists of particles, called *cars*, which move freely according to their velocities. They move within a 2-dimensional space that is divided into evenly sized rectangular cells, called *blocks* (as in city blocks).

The following properties are observed: (1) Cars do not collide with each other, however they do

“bounce” off the boundary edges of the space. (2) Each block counts the number of particles that lie within its boundaries. (3) Each time a particle moves into a new mesh box, the count of the new block increments and the count of the previous block decrements. (4) When the count for a block exceeds a user specified *threshold*, the magnitudes of the velocity of all particles within the block are diminished by a factor *penalty*. (5) If the count drops below *threshold*, the cars' velocities resume their previous magnitudes.

2.2 Problematic Behaviors

The most interesting instances of the simulation are those in which congested areas differ noticeably (in terms of number of cars) from other blocks, and the congested areas move from block to block over time. These will also provide more interesting test cases for the load-balancing scheme, which will have to deal with a dynamic, non-trivial load imbalance. Thus conditions that lead to permanent congestion of some blocks or even distribution of cars are considered problematic. The following is a summary of some of the conditions that cause these problematic behaviors.

Permanent congestion of some blocks has two main causes. The first condition is if cars' velocities are penalized down to zero in congested areas. If fewer than *threshold* cars are stopped within a block, they may still restart if the block's other cars eventually leave, however, if *threshold* or more cars come to a stop within a block, that block is incapable of making progress. The second condition is if *penalty* is too high. Even if no cars are stopped, the congested areas may still be unable to recover in practice.

Even distribution of cars has two main causes. The first is if *threshold* is too low or too high, causing all or none of the blocks to be congested and thus the cars remain relatively evenly distributed. Second, if *penalty* is too low, the congestion in is not sufficiently aggravated by the slowing of cars, causing congested areas differ imperceptibly (in terms of number of cars) from other blocks.

To see that these conditions would cause these problems for extreme values is straightforward. However, the requirements are sometimes conflicting and finding the right balance can be difficult. In the next section, a visualization tool is presented to aid in the understanding of the interactions between the parameters.

2.3 Visualization

To better understand under what values of the various parameters the problematic situations tend to arise, a simple visualization tool was implemented for this problem [12]. The screen shot below shows a sample run with 100 cars.

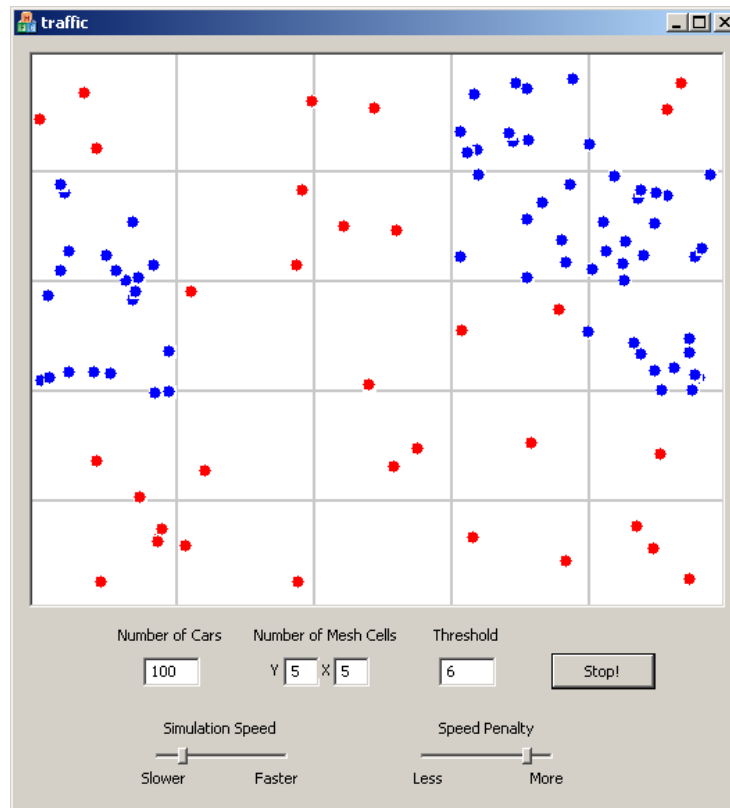


Figure 2: Visualization Tool

The number of cars, the value of *threshold*, and the number of mesh blocks on the X and Y axes are entered in text fields. The value for *penalty* is selected by moving a sliding bar, with a minimum penalty of 0.10 at the “Less” end of the bar, and a maximum value of 0.90 at the “More” end of the bar. In Figure 2, *penalty* is set to 0.76. The “Simulation Speed” sliding bar controls the rate at which the positions are recalculated. This does not affect the actual simulation, only the viewing of it. When the particles are in a congested area, they are colored blue, while the particles in non-congested areas are colored red.

This particular parameter configuration has the desirable behaviors we seek. The average number of cars per non-congested block is 1.68, while the number of cars per congested block is 11.33. Although it can't be seen in the still shot, the location of congested areas is also fairly dynamic.

From watching many different simulations, it appears that *penalty* should be in the range of 0.70 to 0.85 and *thresh* should be approximately 20-30% more than the expected number of cars per block (in an even distribution).

3. Simulator Design

3.1 Cars, Blocks and Car Movement

Each car is represented by an instance of class `Sprite` (the name is from the visualization code, where `sprite` is a moving visual object) and has the following public functions and data members:

```
void Move();
void Slow();
void Fast();
int GetBlock();
int X;
```

```

int          Y;
int          VX;
int          VY;
bool         isSlow;
int          oldVX;
int          oldVY;

```

The variables X and Y are the current position, and $\langle VX, VY \rangle$ is the velocity vector (amount moved per call to Move(), in other words, per timestep). The remaining variables are for convenience in applying and lifting the speed penalty, which is done with the functions Slow() and Fast(). Specifically, oldVX and oldVY hold the “fast” values so they can be restored when the penalty is lifted, since integer multiplication/division are not always reversible. The functions Slow() and Fast() have no effect if the car is already respectively slow or fast. The function GetBlock() returns the block where it is currently located (calculated from X and Y).

Blocks are represented as sets of pointers to cars. (The cars themselves are stored in one large, unordered list.) When a car moves from one block to another, the pointer is moved from one set to the other. At the beginning of each timestep, each block counts its cars, and calls either the Slow() function or the Fast() function on each, depending on whether there are more or fewer than *threshold* of them. Because the Slow() and Fast() functions have no effect if the car is already in that state, the block need not worry what its previous state was, nor which cars have just entered the block and which were there in the previous timestep. This greatly simplifies the programming, though it might be considered inefficient to touch each car instead of only the new or changed ones. After all the blocks have been examined as slow or fast, a second loop calls Move() on each car. This is the end of the current timestep, and the process starts again.

3.2 Data Decomposition and Communication

The unit of work that can be assigned to a processor is a block. Being assigned a block means that that processor is responsible for updating the positions of all the cars currently located in the block. It is not obvious that a space-based decomposition, such as the block, should be used for this particle data. The block level of granularity was chosen because of the nature of the simulation, specifically, because a block needs to count the number of cars within it to determine congestion. If the assignable unit of work were a car, counting the cars within a block might involve a search across many or even all of the processors.

Obviously, cars do not remain in the same block for the duration of the simulation, but constantly move from block to block. Thus efficiently transferring control of a car from one processor to another is at least as important as efficiently updating the car positions within the block. We would like to be able to queue up cars that require sending, and perform several timesteps between the communication phases. Unfortunately, the simulation cannot continue even one extra timestep without knowing whether or not new cars have entered a block. The effects are not only limited to the car(s) that may be missed, but could also affect the behavior of the entire block if the car count is near *threshold*. Thus all the processors are forced to communicate with their neighbors between every timestep.

The loop that updates the cars' positions, which was just described, looks like this:

```

//count the cars in each block and call Slow() or Fast()
...
//move cars
for (carit = CarArray.begin(); carit != CarArray.end(); ++carit) {
    carit->Move();
    if (MyRealBlocks.find(carit->GetBlock()) == MyRealBlocks.end()) {
        node = NodeWithVirtBlock(RealToVirtBlock[carit->GetBlock()]);
    }
}

```

```

        OffNodeCarsSortedByDest[node].push_back(*carit);
        CarArray.erase(carit);
    } else {
        World[realBlock].insert(carit);
        carit++;
    }
}
SendOffNodeCars(OffNodeCarsSortedByDest);

```

First, the car's position is updated with the call to Move(). Then, if the block in which the car now resides is owned by this processor, a pointer to the car is inserted in that block's set. Otherwise, the car is stored (sorted destination processor) in an array of cars that need to be sent to other processors. At the end of the loop, all such cars are sent, one communication per destination processor. (In practice there is a limit to the cars per message, so extremely large groups may be broken up.)

The cars' velocities are constrained such that they cannot move more than one block away in a single timestep, thus each block has 8 neighboring blocks to which cars could move:

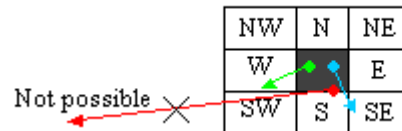


Figure 3: Neighborhood

The neighbors are named according to their map directions, Northwest, North, Northeast, and so forth. All processors have knowledge of which blocks are assigned to all other processors, so they know their neighboring processors. This count will be much less than 8 times the number of blocks on that processor, because a processor's own blocks will share borders, and several neighboring *blocks* may be owned by a single neighboring *processor*. (The assignment of blocks to processors will be discussed in detail in the section on load balancing.) At each timestep, every processor sends, and expects to receive, a list of cars from each of its neighboring processors, although the list may sometimes be empty. The reason is that if a message is only sent when cars need to be sent, a processor cannot know whether a neighbor has no data to send this round, or whether the neighbor is simply running slower and hasn't sent the message yet.

4. Load Balancing Design

4.1 Introduction

The load characteristics of this problem are such that the locations of computation-intensive areas (congested blocks) change over time. Thus optimizing the assignment of blocks ahead of time would not address the problem, and a dynamic approach is needed.

The first step in dynamic load balancing is defining and determining what the load is on a processor. Some approaches (for example [3]) use actual timings to determine the load on a processor. This approach works well when the processors do not perform equally, for example if the hardware is heterogeneous or some processors are busy servicing other tasks. Timings also work well when the load would be difficult or impossible to determine otherwise, for example if the computation is a "black box" that cannot be understood by the load-balancing entity.

For this problem, the load on a processor is defined as the number of cars it is responsible for. This is unrelated to the number of blocks, except that if a block were to be reassigned, the processor would

lose the cars, if any, in that block. The reasons behind this choice are as follows. First, the computation to be performed in each timestep is $O(ncars)$. The communication performed at each timestep is $\alpha * neighbors + \beta * ccars$, where $ccars$ is the number of cars that need to be communicated. We assume $ccars \gg neighbors$, so this is $O(ccars)$. Finally, since Valkyrie's processors [7] are homogenous, simply counting the number of cars seems adequate for estimating the load. One benefit of using the car-based measurement is that it is straightforward to calculate the effect that changes to the block assignment would have on the load (simply add or subtract the cars in reassigned blocks).

There are two major classifications of load-balancing schemes: distributed and global. The distributed approach, notably *diffusion* algorithms, use exchanges with nearby nodes to redistribute the workload without having to use global communication. This approach benefits from lower overhead, but does not have the benefits of a global view. One of the main drawbacks is that for very uneven data, it may take longer to reach a satisfactory distribution of the load.

For this experiment, a global approach was used. This approach ensures that an ideal balance is achieved immediately whenever it is run. This is an important feature for this problem because the congested areas can change drastically very quickly when the threshold is crossed in either direction.

4.2 Block Order

The most effective assignment of blocks to processors won't merely be the one that best balances the number of cars each processor must handle. This is because communication of the off-node cars in each step has an important impact on performance. If blocks from distant ends of the space are assigned together, almost every neighbor will be off-processor. Ideally, we would like to maximize the balance of cars across processors, while minimizing the amount of inter-node communication that is required. However, trying to optimize for both simultaneously would be a very difficult task.

To simplify the optimization, we add the constraint that blocks must be divided into a linear partition, in other words, into contiguous sections. However, using a standard ordering of the blocks, this constraint would result in a decomposition that tends to have long, thin parts with an undesirable amount of surface area. The key is to order the blocks in such a way that blocks next to each other in the ordering are most likely to form low-surface-area squares, and then form a linear partition of the blocks that respects that ordering.

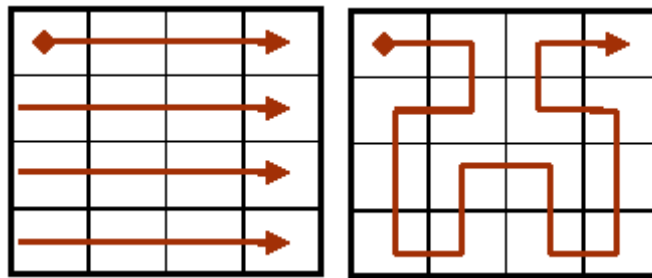


Figure 4: Standard Block Ordering vs. Hilbert Space-Filling Curve (4x4)

The Hilbert Space filling curve [3][4], shown on the right in Figure 4 [5] is just such an ordering. Observe how blocks taken 4 at a time form squares in the figure on the right, whereas they form long rectangles in the figure on the left. For larger numbers of blocks, the Hilbert ordering repeats itself in a larger version of the same pattern, with the pattern shown above as each element (much like a fractal). [3] Experiments using 4x4 and 16x16 configurations are described in the next section.

To implement this improved ordering, the simulator takes a file as input that contains logical indexes in standard order. Thus arbitrary ordering is possible, however the Hilbert ordering was used in all

experiments. For example, the file specifying the 4x4 Hilbert Curve ordering reads:

```

0      1      14      15
3      2      13      12
4      7      8       11
5      6      9       10

```

When calculating the block to which cars belong, the standard numbering is used. This “real block number” is translated to a logical “virtual block number” using the mapping from the file. This mapping is stored in a hash table on every processor.

4.3 Linear Partitioning

The block partition performed at the root node is done using a dynamic programming algorithm for linear partitioning that minimizes the maximum sum over all the partitions. In this case, the algorithm partitions the blocks to minimize the number of cars assigned to the processor assigned the most cars. This is exactly the desired result for load balancing because the execution time for the whole program will be equal to the execution time of the longest running processor. Linear partitioning means that there is a constraint that each partition will have contiguous sections of the input, as mentioned earlier.

Code for the algorithm is as follows [8][9]:

```

//Initialization
p[0] = BlockLoads[0];
for (i=1; i<numBlocks; i++) p[i] = p[i-1] + BlockLoads[i];
for (i=0; i<numBlocks; i++) { M[i+numBlocks*0] = p[i]; D[i+numBlocks*0] =
0; }
for (i=0; i<nodes; i++) { M[0+numBlocks*i] = BlockLoads[0]; D[0+numBlocks*i]
= 0; }

//Compute Optimization
for (i=1; i<numBlocks; i++) {
    for (j=1; j<nodes; j++) {
        int ij = i+numBlocks*j;
        M[ij] = -1;
        for (x=0; x<i; x++) {
            int a = M[x+numBlocks*(j-1)];
            int b = p[i] - p[x];
            int s = a >= b ? a : b;
            if (M[ij] > s || M[ij] == -1) {
                M[ij] = s;
                D[ij] = x;
            }
        }
    }
}

```

The table $M[i,j]$ contains the smallest possible maximum sum for the first i numbers divided into j partitions. New numbers are considered one at a time, using the stored results from earlier computations to quickly get the optimized partition with the new number. This algorithm runs in $O(kn^2)$, where k is the number of partitions (the number of nodes in this case) and n is the number of elements to be partitioned (the number of blocks). The table D contains the locations of the partition boundaries, and it is from D that the actual partitioning will be reconstructed once this loop completes.

4.4 Communication Patterns

Since processors are always assigned contiguous sections of blocks, only an offset and a count are required to specify each assignment. An array of length $2 * \text{nodes}$ stores the assignment of blocks as follows:

- `BlockAssignment[my_rank]` = virtual block offset where my assignment starts
- `BlockAssignment[my_rank + nodes]` = number of blocks, starting at offset, which I am assigned

At the beginning of the simulation, the blocks are divided evenly across the processors, in order of the processor's rank. Every *BalanceCheckFreq* timesteps, each processor sends to the root the count of the number of cars in each of the blocks it is currently assigned. Since processors can have different numbers of blocks, this is achieved with an `MPI_Gatherv` [10][11]:

- ```
MPI_Gatherv(&BlockLoads[BlockAssignment[my_rank]], BlockAssignment[my_rank + nodes], MPI_INT, GBlockLoads, &BlockAssignment[nodes], &BlockAssignment[0], MPI_INT, ROOT, MPI_COMM_WORLD);
```

After this call, the root processor has the car count for every block stored in `GBlockLoads`. The root processor then determines the best partition of the blocks and stores it in *BlockAssignment*. Then a broadcast [10][11] is used to send the assignment to every processor.

- ```
MPI_Bcast(BlockAssignment, 2*nodes, MPI_INT, ROOT, MPI_COMM_WORLD);
```

Unlike the load information, for which each processor only handles information pertaining to its own assignment, every processor needs to have knowledge of the complete partitioning to facilitate both the initial re-shuffling of data for the new assignments, as well as routine communication of off-node cars.

After the new assignments are broadcasted, each processor determines which blocks it no longer owns and their new owners, and the number of new blocks it has been given (the previous owners are not known). Each processor executes the appropriate sends for the data it no longer owns, and performs as many receives as new blocks it has been given. A major inefficiency in this implementation is that the processors do not remember the previous owner of the blocks, so a send and receive is performed for each block. Most likely, many of the blocks being received (or sent) are coming from (or going to) the same processor and could be collapsed into a single message if this fact was known.

5. Results

5.1 Basic Scalability

These experiments were run with no load balancing and a naturally well-balanced load.

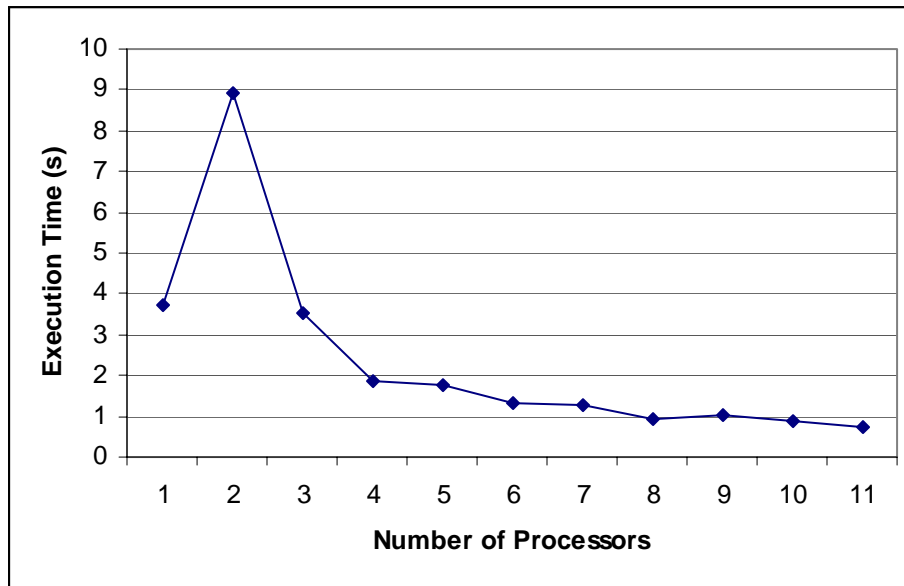


Figure 5: Scaling with Fixed Problem Size (No Load-Balancing)

As shown in Figure 5, execution time takes an enormous hit going from 1 to 2 processors. This is a result of having to communicate the off-processor cars at each timestep, which is not necessary with a single processor. An interesting pattern is seen with 4-5, 6-7, and 8-9 processors; there is a gain in performance going to the even-numbered processor, but no gain in performance (in fact a slowdown from 8-9) going from the even to odd-numbered processor. This is probably a result of the block assignment patterns which result in different numbers of neighbors with which communication must take place.

5.2 Load Balancing Impact on a Balanced Workload

An important property of a load-balancing system is that it does not negatively impact the execution time of workloads that are already balanced. We would like the system to be “invisible” in these cases. While our system generally does not constantly change the block assignments of balanced workloads, there remains the overhead of communicating the load to the root processes and broadcasting the (unchanged) assignments back out to each processor.

A feature that could be added is a minimum threshold of improvement in the balancing. If this threshold is not met, then the root processor will discard the new partition and broadcast the old. This would keep the block assignments from changing too often in workloads that are “close enough” to balanced. In the experiments, it did not appear that the system was having this problem, so the feature was not implemented. This is probably because the granularity of the assignable units (blocks) is large enough that it presents a high enough barrier to change (without being so high that reasonable balance cannot be achieved).

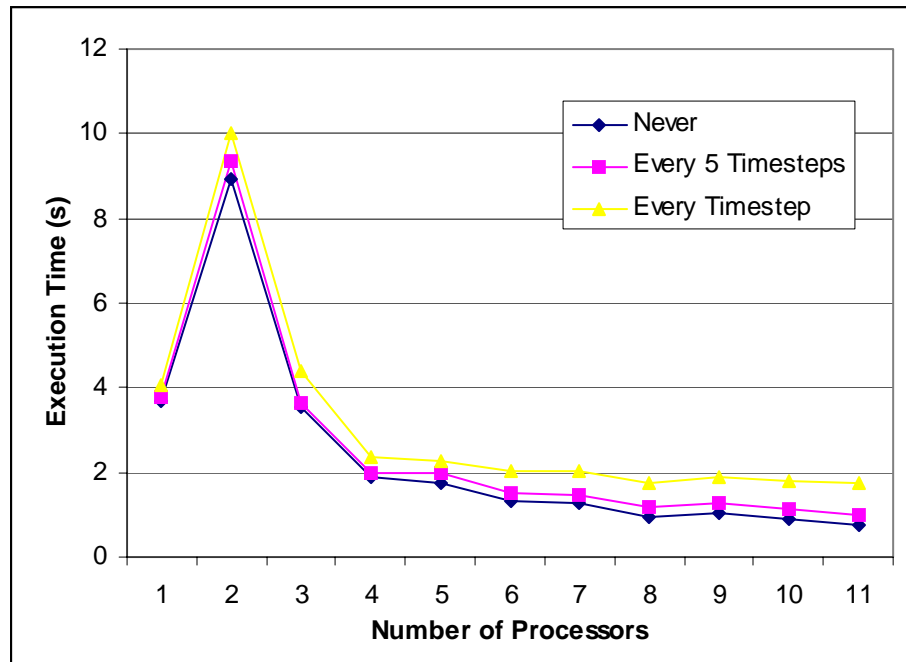


Figure 6: Performance Impact of Load Balancing on Balanced Workload

Figure 6 shows the execution times of experiments with load balancing running every timestep, every 5 timesteps, and never. In practice, every 10 timesteps is probably the most frequently that one would want to run load balancing, so this analysis is in that sense “worst case.” While the load balancing every timestep appears to effectively limit performance to about 2 seconds on this problem, running every 5 timesteps follows ‘never’ case closely, with a small overhead.

5.3 Load Balancing Impact on an Unbalanced Load

Congested areas arise during a well-configured simulation. However, the locations tends to vary randomly, resulting in a situation where *processors* end up with a balanced load, even though some *blocks* are much more heavily congested than others. This happens because each processor is assigned many blocks, some of which are congested and some of which are not, and every processor tends to have the same proportion.

This is different from many standard particle and uneven-load problems, where dense areas tend to occur together. In this simulation, a congested block is not likely to have congested neighbors any more than it is non-congested neighbors.

To force the generation of multi-block neighboring imbalances for the load-balancer to work with, the start condition was changed so that instead of cars starting at a uniform distribution of random points, they all start at either the upper left or lower right corners of the domain.

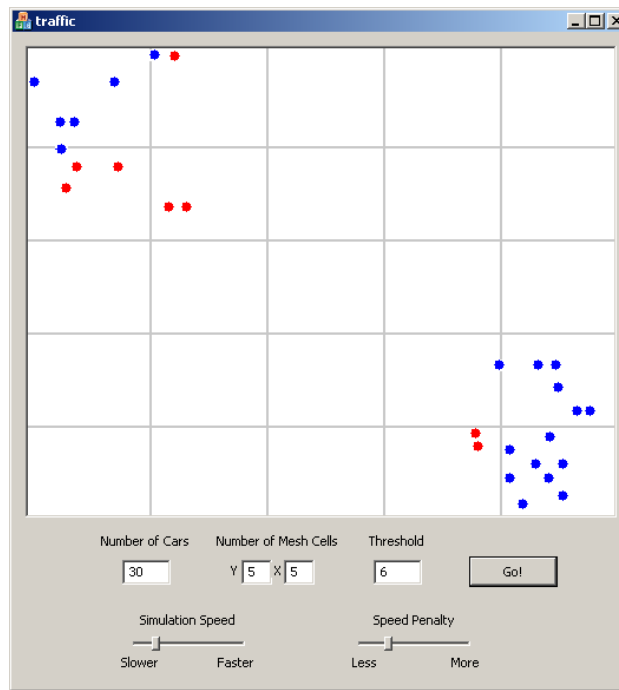


Figure 7: Unbalanced Load Simulation

An example of this modified starting condition is shown in Figure 7, after the first few timesteps. This configuration has the property that the load starts extremely unbalanced, and gets very slowly more balanced. It never gets to a point where it is spread out over more than about 1/3 of the processors during the experiment.

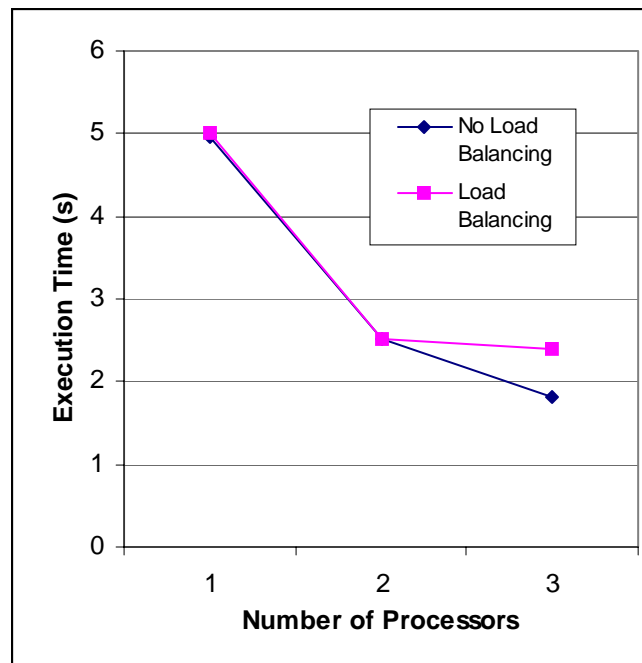


Figure 8: Effect of Load Balancing on Unbalanced Load

As shown in Figure 8, the Load Balancing scheme gets exactly the same execution time on the unbalanced load for 1 and 2 processors. This is in fact a good result, because there are two concentrated areas of work, so for 1 and 2 processors, the natural assignment of blocks to processors matches the one

the balancer would pick, and we can see that the balancer caused no additional overhead (it ran every 300 iterations, out of 3000 total iterations).

Unfortunately, for 3 processors, it caused a slowdown. One problem may be the particular unbalanced workload that I used. From Figures 5 and 6, we see that running with 1 processor is much faster than running with two because no communication is needed for cars that move off-processor. In the unbalanced workload I used, all the heavily loaded blocks fall to one processor for each of the two concentrated corners. While these processors have an unfairly large allotment of work, they can in fact do it very efficiently because no off-processor communication is needed. Also, the load balancer causes more overhead inefficiency in this case than the 1 and 2 processors cases because it actually reassigns blocks, causing the communication of exchanging the associated cars. A workload that is so large that a single processor could not handle an entire concentrated region would probably benefit from the load balancing. Unfortunately, there were technical problems getting such a large workload to run, so there are no results of that kind to report at this time.

6. Comments on Development Process and Future Work

Clearly, the most important unresolved question is what happens with load balancing on a dataset that both requires it and can benefit from it, such as one with clusters of congestion so large that one processor is not capable of handling the cluster alone.

A likely cause of the breakage with large datasets is a problem with the communication facilities. Everything was written from scratch in MPI. I realized only too late how low-level and difficult to work with MPI can be to achieve high-level communication goals. A limitation of MPI also caused the inefficiency in redistributing blocks after a load balancing. The receivers did know their senders; so extra messages were often needed (see Section 4.4). To program around this was complicated, so the inefficiency remained. Again, a high-level goal was frustrated by the low-level details of MPI. Still, in retrospect, I believe this problem was not a small source of inefficiency for the load balancer, and programming around this would have been worth the effort.

Finally, with more time, an exploration of several possible modifications to the simulation would have been an interesting and I believe fruitful investigation. As noted in 5.3, the basic traffic simulation did not result in the kind of *processor* load imbalances that would have been interesting to study. The variation I tried first (described above) turned out to be unfruitful as well, for other reasons. However there are many other interesting possibilities, such as adding barriers through which cars cannot pass, which would create areas of congestion of differing sizes (not just on block boundaries) that would have more complex dissipation patterns.

7. References

[1] For an introduction to ODE solvers, see lecture notes by Scott Baden:

http://www.cse.ucsd.edu/users/baden/classes/cse260_fa02/lectures/Lec06_07/Lec06_07.pdf

[2] Diagrams taken from lecture notes by Scott Baden:

http://www.cse.ucsd.edu/users/baden/classes/cse260_fa02/lectures/Lec14_15/Lec15.pdf.

[3] Frederico D. Sacerdoti, [A Cache-Friendly Liquid Load Balancer](#), M.S. Dissertation, June 2002.

[4] J. R. Pilkington and S. B. Baden. [Dynamic Partitioning of Non-Uniform Structured Workloads with Space filling Curves](#), *IEEE Trans. on Parallel and Distributed Systems*, 7(3), March 1996, pp. 288--300.

[5] Manish Parashar and J.C. Browne. *On partitioning dynamic adaptive grid hierarchies*. In 29th Annual Hawaii International Conference on Systems Sciences, pp. 604-613. Maui, Hawaii, January 1996.

[6] Diagram of Hilbert Space-Filling Curve taken from [3].

[7] The Valkyrie cluster is managed by Academic Computing Services at the University of California, San Diego. <http://www-accs.ucsd.edu/offerings/userhelp/HTML/rocks,d.html>

Introductory references for the Dynamic Programming algorithm for Linear Partitioning:

[8] <http://www.cas.mcmaster.ca/~terlaky/4-6TD3/slides/DP/DP.pdf>

[9] <http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK2/NODE45.HTM>

Introductory references for MPI (Message Passing Interface):

[10] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman Publishers. San Francisco, CA. 1997.

[11] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. *MPI, The Complete Reference*. The MIT Press, Cambridge, MA. 1996. Text available online at: <http://www.netlib.org/utk/papers/mqi-book/mqi-book.html>

8. Acknowledgements

Jusok Lee contributed to writing the basic visual framework of the visualization tool.