CSCI 4/5576: The Random Logistic Map

Amy Le (5576)
Long Tat (4576)
Emily Bertelson (4576)
Kristina Entzel (4576)

December 16, 2014

1 Abstract

The purpose of this investigation was to explore and simulate a spatially random logistic map using a dynamic load balancer on Janus, the CU supercomputer [5]. The recursive nature of the map prevents the individual fixed point iterations from being parallelized, but a set of iterations may be load balanced over many cores. The original simulation was written in serial code in MATLAB. Our solution has modified the original version for efficiency, speed, and scalability. We implemented our simulation in C++ and present our results in terms of a histogram of observed periodic orbits and a bifurcation diagram. Single core optimization techniques, such as SIMD loop vectorization and function inlining, as well as using a dynamic load balancer for more efficient work distribution were applied. The HDF5 file format was used to store the simulation results in a better archival format. The benchmarking (weak scaling study) results imply the best speedup and efficiency is gained when invoking the load balancer on one node (12 cores), although we tested our simulation over 16 nodes (192 cores). Improvements to this project include optimizing the post-simulation data processing.

2 Introduction

2.1 Deterministic case

The Logistic map is a quadratic recursive equation on the domain [0,1]. It is a popularly studied topic in nonlinear dynamics and has applications in population modeling. There is one parameter in the expression, r, which can take any value in the range [0,4].

$$x_{n+1} = r x_n (1 - x_n)$$

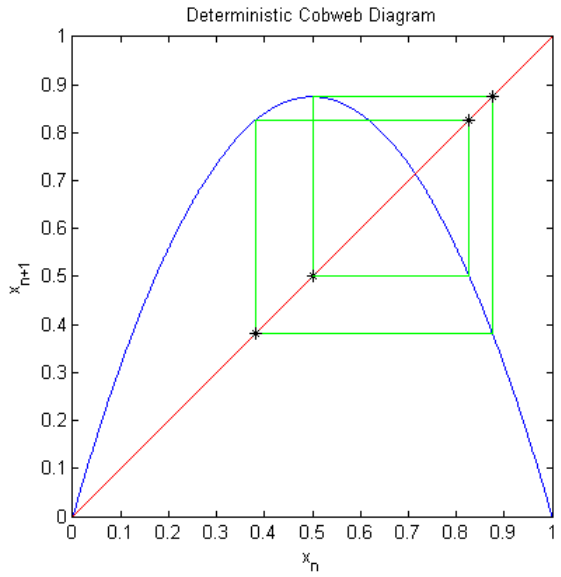

Figure 1: Deterministic Logistic Map (blue) for $r = 3.2$. There is a stable period 4 orbit. The order of the period is calculated by counting the number of crossings (green) on the line $x_{n+1} = x_n$ (red). Between $r \in [0, 3.5]$, we observe stable periodic orbits.

0rp

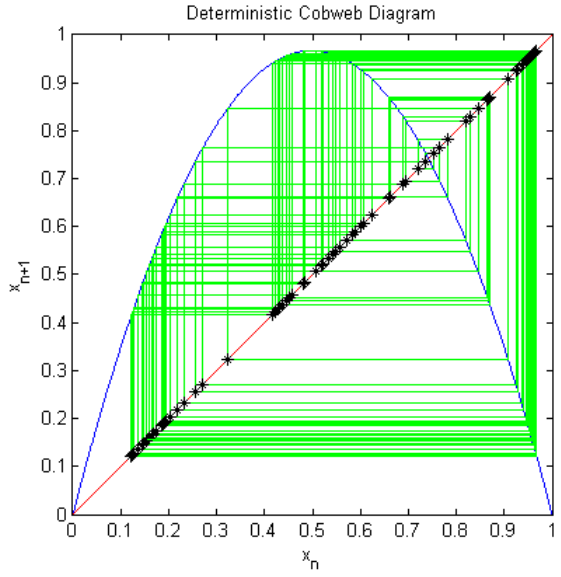

Figure 2: Deterministic Logistic Map (blue) for $r = 3.8$. There is a no stable orbit. For values of $r \in [3.5, 4]$, the system experiences the onset of chaos.

There are two ways to vary the deterministic map. We can simulate the parameter r as a function of time or space. The existing literature explore the notion of randomness in time, so we explore r as a function of space [1].

2.2 Random case

The following equation is the fixed point iteration that the code completes, where $R(x)$ is calculated by manipulating a random number generator.

$$x_{n+1} = R(x_n)x_n(1 - x_n)$$

The exact details of how to calculate $R(x)$ are outlined below.

$$\begin{aligned}
\ln(R(x)) &= \xi(x) \\
\xi(x) &= \ln(r) + 2\sum_{n=1}^{N} a_n \cos(2\pi n x) - b_n \sin(2\pi n x) \\
a_n, b_n &\sim Unif(-M_n, M_n) \\
M_n &= \sqrt{1.5 S_n} \\
S_n &= \alpha e^{-L|n|} \\
\alpha &= \sigma^2 \tanh(L/2) \\
\sigma &< \ln(4/r) \frac{\tanh(L/4)}{\sqrt{1.5 \tanh(L/2)}}
\end{aligned}$$

Where $L \in (0, 1)$ represents the correlation length (and is fixed for each simulation) and $r \in [0, 4]$ is also fixed for each simulation.

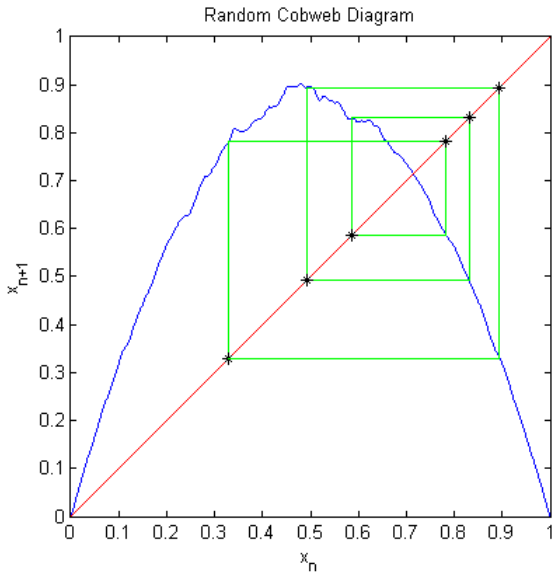

Figure 3: One instance of a random logistic map (blue). The map has converged to a stable period 6 orbit (green). Notice the "wiggliness" in the parabola shape.

Other instances of the random map would vary from this realization, due to the random nature of the map.

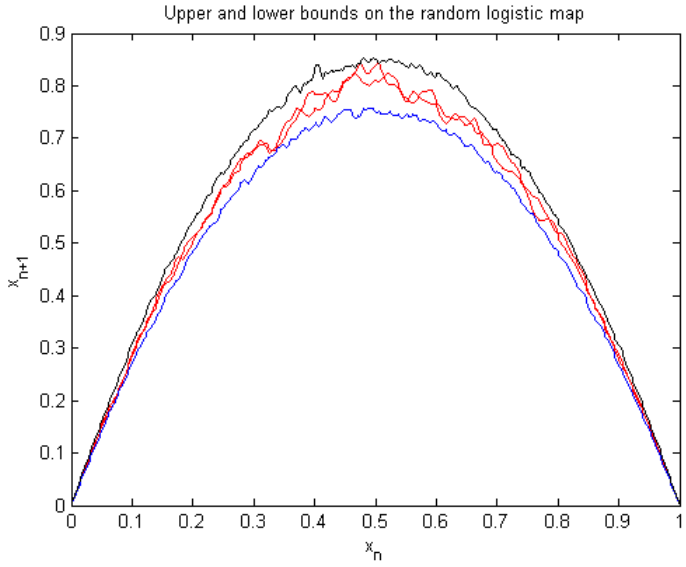

Figure 4: A coarse demonstration of the upper (black) and lower (blue) bounds of the random logistic map. Sample realizations are shown in red.

2.3 Project Goals

Since the map can take on a range of values for any given position in space, it would be useful to characterize some of its properties. In particular, we will be studying the stability of the map, which includes locating fixed points and generating bifurcation diagrams. The two main goals are:

1. Find the expected number of order p periodic orbits for a the random map ($p = 1, 2, 3, ...$)
   (a) For an initial starting value $x_0 \in [0, 1]$ and a specific random function $R_0(x)$, iterate until you find the fixed point(s), x_i^* associated with $R_0(x)$.
   (b) Classify the fixed points in terms of a period p orbit.
   (c) Each processor should take a different initial x_0 and report whether the initial condition led to finding a unique stable orbit
      - The processors should be properly load balanced.
      - As each processor finishes its work, it will write its results to an HDF5 file (parallel i/o)
   (d) Repeat the above steps for a large number of different random maps $R_i(x)$, $i = 0, 1, 2, ...\bar{N}$ in order to find the expected number of order p periodic orbits for the random map.
2. Create a set valued bifurcation diagram [3]
   (a) For many values of $r \in [0, 4]$, and a fixed random function $R_0(x)$, plot the locations of the periodic orbits as a function of r. A period p orbit will have p corresponding x values as its orbit locations (e.g. a period 1 orbit will have 1 fixed point, a period 2 orbit will have 2 fixed points, and so on).
      - Use a HDF5 file to store the simulation data and as a source to generate bifurcation diagrams

As the map has an element of randomness to it, many simulations (a large $\bar{N}$) would be required for statistical analysis.

3 Method

After profiling the MATLAB code, we realized that our critical path is the fixed point iteration calculation, so parallelizing our solver is the necessary step to improve our performance. Unlike the problems we encountered in class, we did not have a matrix or an array which we can decompose and distribute to multiple processors. Our solver uses the previous calculation to compute the current calculation. This dependency renders parallelization with OpenMP/MPI impossible.

Upon carefully reviewing our objective, we noticed that this project was originally designed to collect data over a range of inputs. If parallelizing at the file level is impossible, we

parallelize the project at the simulation level. This is possible because the fixed point iterations are independent over a range of initial conditions. A load balancer suits our purposes perfectly because the load balancer will redistribute work over many processors to eliminate wait time. The general progression of the project is outlined below:

1. Convert the MATLAB code to C++
2. Confirm the C++ versions of the code that we each produce work together correctly by comparing to the serial version
3. Invoke the load balancer to assign an initial condition to each fixed point iteration
4. Benchmark: strong scaling study (speedup and efficiency)

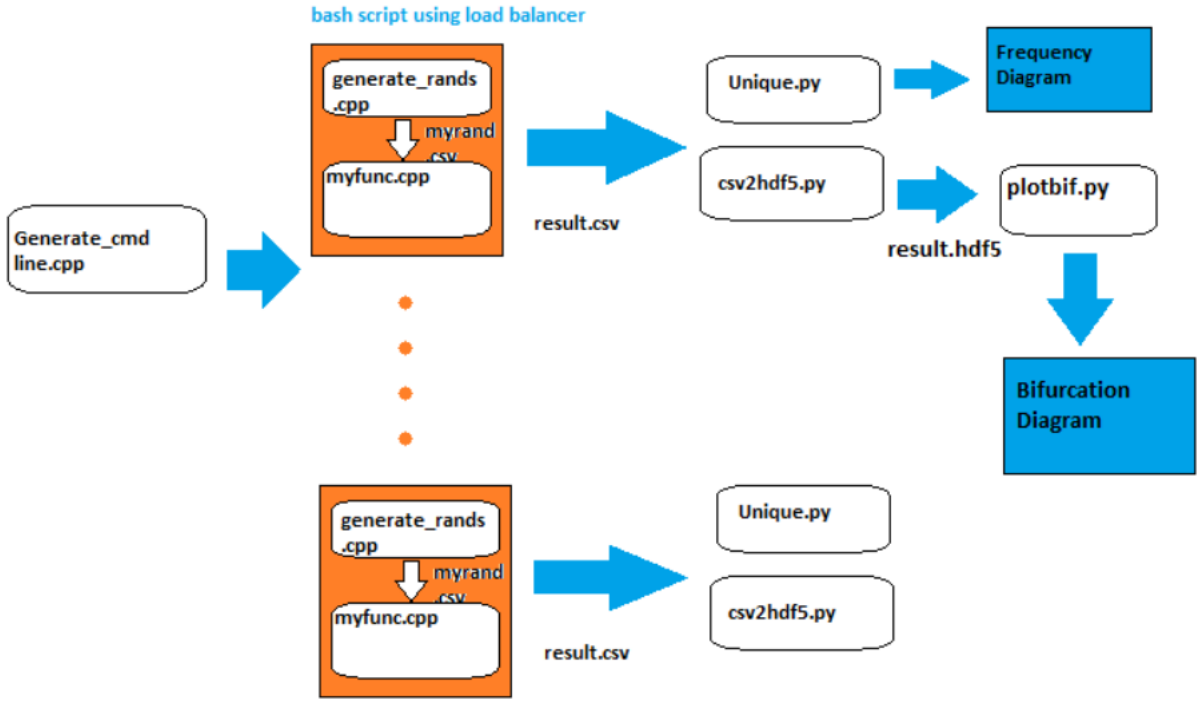

Figure 5: Workflow

The Workflow (Figure 5) demonstrates that the program begins with `generate_cmdlines.exe`. We start with `generate_cmdlines.exe` file which is the set up file for our simulation.

1. `generate_cmdlines.exe`: This file will take 4 inputs (L,dr,dx,iter). It will generate all the bash script files.
2. Bash Script files: Each of these scripts will invoke `generate_rands.exe` once which will take in with the same L given in `generate_cmdlines.exe`, an r value which is uniformly distributed between 0 and 4 with step size dr and an output file. This outfile

contains the randomized parameters a, b, used by `myfunc.exe`. This script also invokes the load balancer which will distribute tasks over the available processors[1].

3. `generate_rands.exe`: Generate randomized parameters a, b and put it into an output file for given L and r values.

4. `myfunc.exe`: Run the fixed point iteration and print out the orbit locations if they exist or return nothing if the map diverges. All output from `myfunc.exe` will be fed into a `result.csv` associated with that bash script file. Note that we can concatenate all these result files if we want to study the map's behavior as a whole.

5. `csv2hdf5.py`: Convert `result.csv` to a HDF5 file while using `unique.py` to check for uniqueness in the data set. Save the processed data in an HDF5 file for archival purposes. `unique.py` also creates a histogram of the data.

6. `plotbif.py`: Use the HDF5 file to produce the bifurcation diagram.

3.1 Original Serial Implementation in MATLAB

Since MATLAB is inherently slower than C++, we expect seeing the code speedup simply from porting the code to C++. Besides being written in serial, the code was slow due to some inefficient I/O practices. After running a fixed point iteration, the code would create a file and write the results to it. In other words, for every fixed point iteration, that corresponds with an initial condition x_0 and a parameter value r, a new file is created. Furthermore, the data written was not checked until after all files were created for uniqueness and convergence. Diverging cases where no period orbit was found would create a file full of junk data. The data processing steps included reopening all the data files, removing duplicates, and sorting.

To solve these practices we processed the data before writing to a single csv file to avoid writing diverging orbits to the file. To check uniqueness, we opened the csv file once, and checked line by line (each line is a set of data) for uniqueness while comparing to an array (in memory) that kept track of the unique data points. If the data was unique, it was added to the array. These practices resulted in a smaller set of data that was more manageable to graph.

3.2 Single Core Optimization

Optimizations implemented in the code conversion:

- Preferential use of the multiply and add operators where possible, since they are less expensive than subtract and divide operators

- Used a reduction on the loop that computes the Fourier Series in order to take advantage of the data parallelism with SIMD

[1]Each task corresponds to executing `myfunc.exe` with a unique set of parameters: L, r, x_0, a, b. Different initial conditions x_0 are uniformly distributed between 0 and 1 with step size dx

- Loop structure was reorganized to take advantage of C++ being row-oriented (outer loop should go by rows, then inner loops go by columns)
- Inlining in the C++ code to reduce the number of function calls
- The lack of a built in uniform random number generator that generates a random double between two doubles led to the creation of a psudeo random number generator with the use of `rand` and `srand`.

```
/*Produce a random number in the range [a,b]*/
double rand_draw(double a, double b) {
  double random = ((double) rand()) / (double) RAND_MAX;
  double diff = b - a;
  double r = random * diff;
  return a + r;
}
```

3.3 Load Balancer

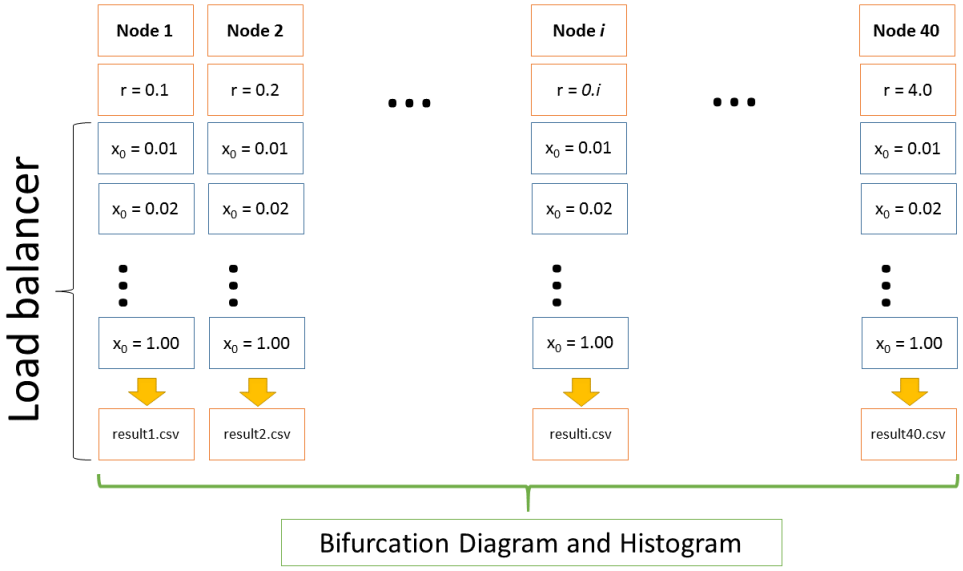

Figure 6: A load balancer recognizes the number of processors that are going to be used and manages the workload distribution among them. It first assigns all processors a chunk of work, and when a processor finishes the load balancer assigns that processor more work.

We researched types of load balancers [4]. We found there are many strategies for load balancing, such as sender initiated diffusion, receiver initiated diffusion, hierarchical balance model, etc. [7]. This investigation used the Load Balance tool on Janus, which invokes a master-slave strategy for balancing [5].

During fixed point iteration, we take a starting value, perform initial calculations, and then use this guess solution to find the next guess, and so on. Each iteration's solution is dependent on the previous iteration's solution, so this calculating the fixed period orbit for a starting point must be done in serial. Some starting values will converge to a fixed period orbit much more quickly than others, and some starting values will diverge, and never find a fixed period orbit at all. The fixed point iteration for a starting value is completely independent from another starting value, so we can parallelize this by assigning each processor a starting value to process. This allows us to optimize despite having a variety of runtimes.

First we assign a global r to a node, then a starting value x_0 to each processor. The diverging calculations will take place on processors in parallel to converging calculations on other processors and no time is wasted waiting. When a processor has either found a solution or run through the 1000 iterations[2] it stops the calculation and writes the resulting fixed period orbits to a csv for that node.

3.3.1 Write Locks

Originally we had a single csv file for the entire program where results would be written to. We ran simulations with different initial r and L values across many nodes and wrote all the fixed period orbits into a single csv file along with the accompanying simulation parameters, r and L. This practiced caused unexpected I/O problems. The load balancer manages the workload of all processors specified in slurm's `#SBATCH -N` parameter. However, when we used many different instances of the load balancer (invoking the load balancer on different sets of jobs, each one independent of the other), we observed erratic results due to multiple processors from different jobs trying to write to the same file. This means the load balancer tool invoked on independent jobs did not communicate between the jobs, so upon finding a fixed period orbit, would write to the csv file with no consideration of synchronization. The resulting csv was an unreadable jumble of partial lines of solutions from different nodes[3]. One possible solution is to implement a system of write locks, so that only one processor from any given run may write to the file at a time. However, this has the potential to create a bottleneck for our program. Therefore, we pursued a different strategy.

We resolved this issue by assigning a unique result file to each instance that we called the load balancer on. Even though we ended up creating as many files as we had nodes, this is still a massive improvement in the I/O and number of files created from the original implementation. Our final algorithm created N files over N nodes, whereas the original implementation created $1000 * N$ files over N values of r and 1000 initial conditions.

3.4 HDF5

While the output was initially stored in a comma-separated value format, the HDF5 file format was chosen for final storage of the data. An intermediary step (`unique.py`) removes redundancies from the .csv file before copying the data into an HDF5 file (via `csv2hdf5.py`). Two resources were used to implement the HDF5 aspect of the project [2] [6].

[2]The ceiling number of iterations we applied to any given initial condition is 1000 iterates.

[3]Upon further reflection, this seems like a logical consequence of calling the load balancer over independent runs.

The hierarchical structures allowed by the HDF5 format were ideal for our uses. We developed a structure with the intention of making the data convenient and efficient to access for the production of a bifurcation diagram and histogram, as well as future analysis.

The folowing is the structure of the output HDF5 file. Two groups within the root store the data twice, in different formats; group `arch` is used for plotting histograms, while group `bif` stores three-dimensional coordinates for plotting bifurcation diagrams.

`arch`
- group "r"
  - group "L"
    - group "$p = 1$"
      - dataset
        - (x_1)
        - (x_2)
        - (x_3)
        - ...
    - group "$p = 2$"
      - dataset
        - (x_{11}, x_{12})
        - (x_{21}, x_{22})
        - ...

`bif`
- group "L"
  - dataset
    - $(x_1, r, p = 1)$
    - $(x_1, x_2, r, p = 2)$
    - ...
    - $(x_1, x_2, ...x_k, r, p = k)$

4 Results

The results of the two sets of data give us the following histograms and bifurcation diagrams. If our implementation is correct, the results using the load balancer and C++ will strongly resemble the results from the original Matlab code. The nature of the equation makes the exact replication of a single run difficult – the problem involves random noise. Before implementing the ability to apply extreme constraints to the parameters, we could not replicate a single run, but preliminary results appeared as expected and were accepted as correct. Further work after the presentation hardcoding the random coefficients into the C++ and Matlab code resulted in identical output and tested the correctness of our converted code.

Results from running the program in Matlab are slightly different for each simulation but seem follow a general pattern, which is the subject of the study.

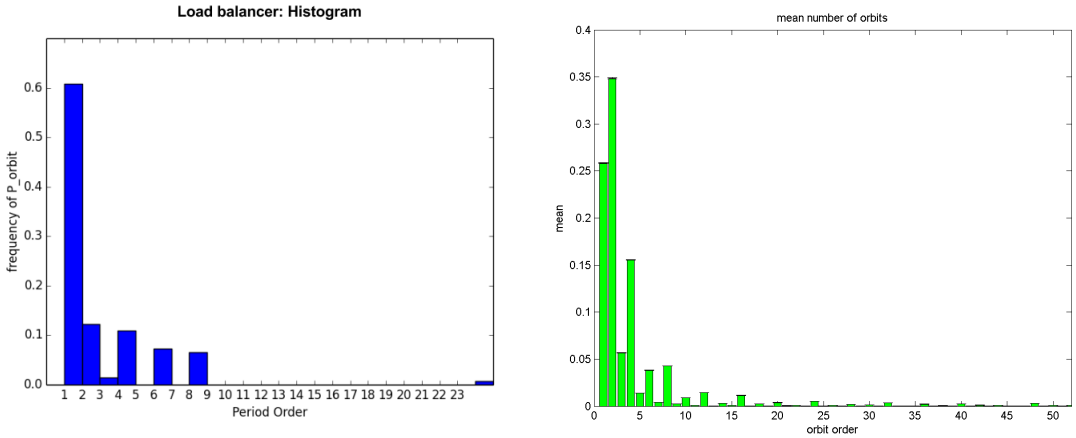

Table 1: Load balanced histogram (left) and Serial histogram (right). The histograms display the frequency of different period order orbits for given r and L values. They both indicate a tendency toward a high frequency of low order period orbits with few high order outliers.

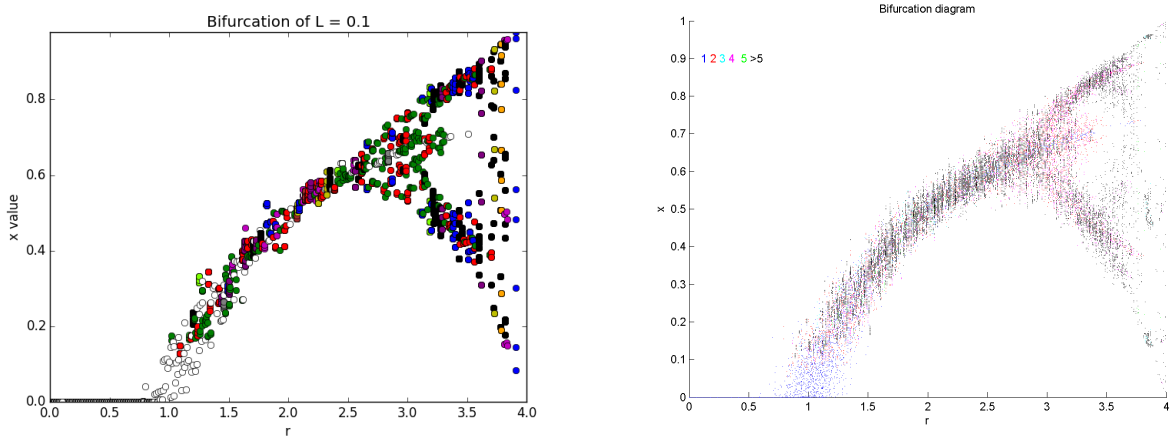

Table 2: Load balanced Bifurcation with $L = 0.1$ (left) and Serial Bifurcation with $L = 0.1$ (right). The bifurcation diagrams plot the variable r versus the resulting x values for a given L. We observe bifurcations from the load balanced implementation occurring in similar places as the serial implementation.

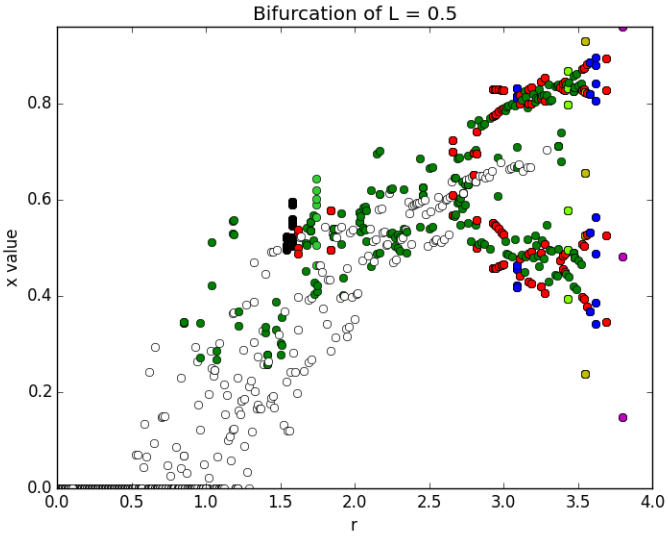

Figure 7: Bifurcation with $L = 0.5$

4.1 Benchmarking

The original code written in MATLAB took hours to run a simulation, and the optimized version of the code with the load balancer ran in approximately 30 minutes. The benchmarking was computed using the C++ code for homogeneity. We tested the code over 16 nodes (192 processors).

Running the simulation to generate a bifurcation diagram requires choosing a number of r values you want to graph and running the fixed point iteration on a set of values for that r and an associated set of random coefficients. This means executing `generate_rands.exe` for each r to get the correct coefficients. The fixed problem size for benchmarking was testing 40 r values between [0,4], and 1000 initial conditions x_0 in [0,1] for each value of r.

For the serial run, `generate_rands.exe` was called 40 times and each initial condition executed serially. For the parallel cases we increase the number of nodes used (from 1 node to 16 nodes), run `generate_rands.exe` over each value of r and call the load balancer which will distribute the work among the increased number of processors.

We found that the best Speedup and Efficiency occurred for one node (12 processors), though we may expect better Speedup and performance with more r values tested (perhaps on the order of 100 instead of 40). It is possible the communication overhead was high in proportion to the amount of work given to each processor due to only finding the solution or 40 values of r.

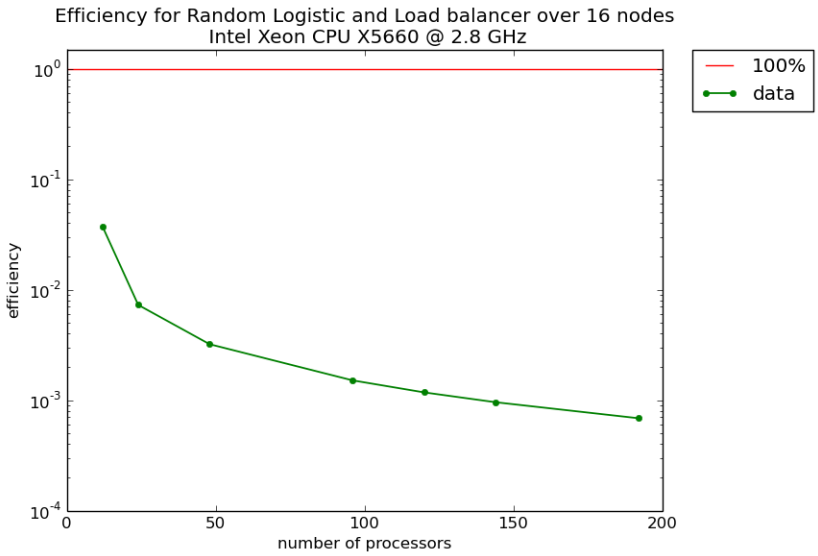

Figure 8: Efficiency of our code. We found that the best Efficiency occurred for one node (12 processors).

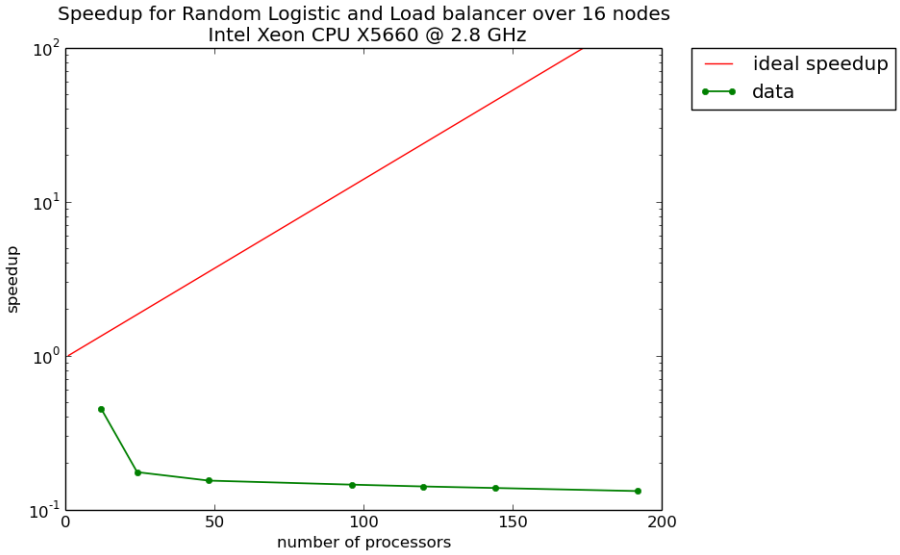

Figure 9: Speedup of our code. We found that the best Speedup occurred for one node (12 processors).

5 Conclusion

Overall, the results of this investigation were positive. Since the serial and load balanced versions of the simulation fall in relatively the same range of values (Table 1, and Table 2), we believe the load-balanced versions output the accurate results. The histograms give

us an idea of the sort of periodic orbits that occur most frequently in the random logistic map. Table 1 shows that the most commonly observed orbit is period 1. The bifurcation diagrams demonstrate the periodic orbits are highly affected by the correlation length L in the simulation. It seems that for larger values of L, eg. $L = 0.5$ (Figure 7), there are fewer high order periodic orbits than for lower values of L, eg. $L = 0.1$ (Table 2). This is indicated by the larger density of white circles on the plot (white circles represent period 1).

5.1 Future Work

5.1.1 Post-simulation processing

Our group focused more on the single-core optimization and dynamic load balancing aspects of the simulation than on how to most efficiently process the simulation data for generating the bifurcation diagram and histogram. Some improvements we could work on in the future would be to explore how to best remove duplicate orbits (as more than one initial condition can converge to the same periodic orbit) as we write the data to the HDF5 file. Another improvement would be to use the link properties of the HDF5 file format to link the data for each bifurcation diagram together, making a new "view" of the data for each kind of bifurcation diagram.

5.1.2 Scaling Study

It was surprising that the speedup (Figure 9) plot showed the optimal number of nodes is one. We suspect this may be due to the fixed problem size we assigned to each set of nodes. Perhaps if the problem size (number of initial conditions x_0 = number of tasks for the load balancer) were larger, we would see a speedup graph that increases over some nodes and then plateaus for too many nodes. This investigation used 1,000 tasks, but a future scaling study would increase the number of tasks to about 10,000 or more.

References

[1] Athreya, K. and Dai, J. (2000). Random logistic maps. *Journal of Theoretical Probability*, 13(2):595–608.

[2] Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, New York, NY, USA. ACM.

[3] Lamb, J. S., Rasmussen, M., and Rodrigues, C. S. (2013). Topological bifurcations of minimal invariant sets for set-valued dynamical systems. *Proceedings of the American Mathematical Society.*

[4] Olivier, S. and Prins, J. (2008). Scalable dynamic load balancing using UPC. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 123–131.

[5] Research Computing at University of Colorado Boulder (2014). Load balance tool. https://www.rc.colorado.edu/node/507.

[6] The HDF Group (2014). Hdf5 user's guide. http://www.hdfgroup.org/HDF5/doc/UG/index.html.

[7] Willibeek-LeMair, M. H. and Reeves, A. P. (1993). Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Computing*, 4(4).