Efficient Biased Random Bit Generation for Parallel Processing

Dale M. Slone
(M.S. Thesis)

September 28, 1994
DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.
Efficient Biased Random Bit Generation for Parallel Processing

Dale M. Slone
(M.S. Thesis)

Manuscript date: September 28, 1994
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Efficient Biased Random Bit Generation For Parallel Processing

By

Dale M. Slone
B.S. (California State University, Stanislaus) 1990

THESIS

Submitted in partial satisfaction of the requirements for the degree of
MASTER OF SCIENCE
in
Computer Science
in the
GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

[Signatures]

Committee in Charge

1994
Preface

This thesis work grew out of an idea from Garry Rodrigue’s Numerical Solutions for Partial Differential Equations course. Lattice gas methods were mentioned in passing, and I was intrigued by the idea of simulations which used bit and integer operations rather than the usual floating point techniques. Lattice gases seemed to be the ideal application with which to learn parallel processing. Overcoming the problems associated with generating random numbers in parallel turned out to be a non-trivial problem, to which this thesis addresses itself.

The thesis was prepared using the \TeX\ and \LaTeX\ document preparation system. Plots were created using the freeware package gnuplot and the paint program MacDraw.

I wish to thank the Lawrence Livermore National Laboratory for their financial support through the Student-Employee program and also for allowing me access to many costly and therefore fun computers. General thanks also to the support and consulting staffs of both NERSC and the MPCI at the lab for their kind and generous help.

Thanks to Garry for letting me run with this project and entry into the wonders of parallel processing. I appreciate the help and guidance he gave along the way. Thanks to Guy and Bracy for their time and many suggestions regarding lattice gases and random numbers. Thanks to the many members of the Internet community who took the time to answer even my most basic questions and point me along the one true way. A special thanks to my committee who read through the thesis and offered their suggestions without once suggesting it was wasted effort.

Thanks also to my parents, Murray and Sylvia, for their constant encouragement and to my brother, Dan, who introduced me to so much.

Further thanks to Al, Cindy, Doug, Debbie, and Christine for slogging along and understanding the ups and downs. Thanks also to Ed, Scott, and Andy who accepted it all
without really understanding. Thanks to Paul for his many TeX hints and always interesting lunchtime discussions. Special thanx to the girls, who still love me, and especially to Kim, who kept me going and pushed me when I needed it and hugged me when I really needed it. Rocket.
Contents

Preface ii

List of Tables vi

List of Figures vii

1 Introduction 1

2 Random Number Generation 7
  2.1 Introduction ............................................ 7
  2.2 Pseudo-Random Number Generators ...................... 8
    2.2.1 Multiplicative Congruential Generators .......... 9
    2.2.2 Shift Register Generators ......................... 11
    2.2.3 Lagged Fibonacci Generators ....................... 12
    2.2.4 Cellular Automata Generators ..................... 14
  2.3 Statistical Measurements of Generators ............... 16
    2.3.1 Average Properties .................................. 18
    2.3.2 Ordered and Unordered \( \chi^2 \) Tests .......... 19
    2.3.3 Distributional \( \chi^2 \) Tests .................. 23
  2.4 Random Bit Generation and Testing .................... 26
    2.4.1 A Random Bit Generator ........................... 26
    2.4.2 \( m \)-tuple Test ................................. 28
    2.4.3 Biasing the Bits ................................... 30
  2.5 Parallel Considerations .............................. 34
3 Lattice Gas Automata

3.1 Introduction ........................................ 37
3.2 An Automaton for Burgers' Equation .................... 39
3.3 A Bit Vector Implementation .......................... 43

4 Parallel Platforms and Results ........................ 52
4.1 Sequential Results .................................. 52
4.2 Parallel Considerations .............................. 56
4.3 The BBN TC2000 .................................. 58
   4.3.1 The Split-Join Model ............................ 60
   4.3.1.1 Single Node Results ........................ 62
   4.3.1.2 Multi-Processor Results .................... 62
   4.3.2 The Message Passing Model ....................... 63
   4.3.2.1 Single Node Results ........................ 66
   4.3.2.2 Multi-Processor Results .................... 66
4.4 Multitasking on the CRAY C90 ........................ 67
   4.4.1 Single Node Results .......................... 70
   4.4.2 Multi-Processor Results ....................... 70

5 Conclusions .................................... 73
5.1 Summary ........................................ 73
5.2 Future Work ...................................... 76

A Chapman-Enskog Expansion .......................... 79

Bibliography ........................................ 85
List of Tables

2.1 Average properties of Multiplicative Congruential generators .................. 10
2.2 Average properties of Shift Register generators ...................................... 12
2.3 Average properties of Lagged Fibonacci generators .............................. 13
2.4 Lookup table for rule 30, \(k = 2, r = 1\) automaton ................................. 15
2.5 Average properties of Cellular Automaton generators ............................ 16
2.6 Table of unordered subinterval frequencies for \(LF[55,24,32,+]\) .................. 22
2.7 Table of ordered subinterval frequencies for \(LF[55,24,32,+]\) .................. 23
2.8 \(\chi^2\) tests of \(LF[55,24,32,+]\) random number generator ....................... 23
2.9 Distributional tests of \(LF[55,24,32,+]\) random number generator ............... 25
2.10 Number of sequential bits needed for biasing .................................... 31
3.1 Collision rules for Burgers' equation lattice gas .................................. 41
3.2 Table of parameters for Burgers' equation lattice gas ............................. 42
4.1 Lattice gas execution profile percentages ........................................... 53
4.2 Execution times on BBN TC2000 using PCP .......................................... 63
4.3 Execution times on BBN TC2000 using LMPS ........................................ 67
4.4 Execution times on CRAY C90 ............................................................. 71
A.1 Collision rules for Burgers' equation lattice gas .................................... 79
List of Figures

2.1 LF[55,24,32,+] vs CA30 random number generation times .................. 17
2.2 A visually good generator. MC[60,0,503] ..................................... 18
2.3 A visually bad generator. MC[62,0,503] .................................... 19
2.4 LF[55,24,32,+] vs (30,6,4,1,0) random bit generation times .......... 27
2.5 Time needed to bias $10^6$ bits ............................................ 32
2.6 Horizontal distribution of random bits ...................................... 33
2.7 Vertical distribution of random bits ........................................ 34
3.1 Grid for Burgers’ equation lattice gas ...................................... 39
3.2 Valid particle states ............................................................ 40
3.3 Collision Rules ................................................................. 40
3.4 Initial Condition ............................................................... 42
3.5 Solution at 0.25 sec ........................................................... 43
3.6 Initial condition evolution as the aggregate of grids increases ........ 51
4.1 Initial condition, 256 Sites .................................................. 53
4.2 Solution at 0.25 sec, 256 Sites .............................................. 54
4.3 Initial condition, 16384 Sites ............................................... 55
4.4 Solution at 0.25 sec, 16384 Sites ........................................... 55
4.5 32-node butterfly architecture ............................................... 59
4.6 16-processor BBN butterfly architecture .................................. 60
4.7 Single processor execution times on BBN TC2000 using PCP .......... 62
4.8 Multi-processor execution times on BBN TC2000 using PCP .......... 64
4.9 Single processor execution times on BBN TC2000 using LMPS ........ 66
4.10 Multi-processor execution times on BBN TC2000 using LMPS ........ 68
4.11 Single processor execution times on CRAY C90 .......................... 70
4.12 Multi-processor execution times on CRAY C90 .......................... 72
Chapter 1

Introduction

The generation of random numbers is an important aspect of scientific computing. Calculations that are inherently probabilistic, such as photon transport [ZJ89, FHL87, WABK91], Monte Carlo Integration [KW86], and Ising model simulation [PdO90] all require that long sequences of random numbers be generated. Statistical applications, such as estimating distributional functions, comparing statistical procedures, geometric probability, and generating and testing for large primes in encryption schemes, all require long streams of random numbers. In addition, statistical methods called lattice gases have recently been developed to model systems that behave according to the solutions of systems of diffusive partial differential equations, such as those governed by the Navier-Stokes equations [dLF86, FdH87, FHP86]. Lattice gases have been used to model differential equations such as the diffusion equation [LB90], heat equation [Rod86], and the wave equation [CCDL88]. Lattice gas methods have also been used to examine more complex systems such as flow in porous media [Rot88, Pap93], advection-diffusion equations [EL93, BB92, BB91], magnetohydrodynamics [CM87], electrostatics [MD87], and immiscible two-phase fluids [RK88]. They represent an alternative to standard finite difference method schemes and have been applied to problems with complex geometries such as the equilibrium and growth morphology of a crystal in a diffusion field [SU90], aerosol droplet coagulation and deformation [DSF91], and transport properties of tracers in a Coulomb gas [PY91]. The stability and Boolean structure of lattice gases is well-suited to other particle transport models such
as electron transport and degeneracy effects in semiconductor devices [Anc90]. Extending lattice gases to regions of high-Reynolds-number flow has recently been suggested as a technique to perform large-eddy simulation of turbulent flow [KO93].

Much theoretical work has gone into techniques for analyzing the statistical measures of the randomness of the generated random numbers and also the periodicity and short-term structure of various random number generators [Knu81, Nie87, HD62, TT56, ADG70]. For large-scale computations run on parallel machines and requiring billions of random numbers, the standard library random number generators have too short a period to be considered random and the numbers generated may not be random either in a bit-wise sense or in a sequential statistical sense [APG92]. In addition, care must be taken to ensure that correlations aren’t introduced as the numbers are being generated in parallel. Ideally, random numbers would be generated as a result of a physical process, not a computational procedure. They would be independent of the application being computed and truly random in the sense of unpredictability and nondeterminism. Any such method would require extraneous hardware, such as position-sensitive photo-optics [MM91], to generate a single random number and is much too slow to generate the billions of random numbers needed in typical parallel computations. Software must therefore be used to create the sequence of values needed.

However, for some applications, a random number generator which generates only a small subrange of all the possible random numbers is necessary. For example, only integers in the interval \([0, 9]\) might need to be generated when the machine can generate integers in the (much larger) interval \([0, 2^{15} - 1]\). Large discrepancies between the interval of interest and the interval of random number generation lead to a waste of space and computational effort. Lattice gases are an entire class of computations in which random numbers are required over only a partial range of potential values. Lattice gases are cellular automata in which particles move on a lattice according to a given set of collision and advection rules, which may be probabilistic. For each direction on the lattice grid, one bit is needed at each site to indicate the presence or absence of a particle. Particle properties also require bits to signify whether a particle has, or doesn’t have, a particular property.

Cellular automata are a class of mathematical systems which are characterized by discreteness in space, time, and independent variable. Restricting the automata to a regular
lattice guarantees spatial discretization. Evolving the automata in discrete time steps enforces the temporal discretization. Representing the state value by a few bits at each lattice site ensures the discretization of the independent variable. The power of cellular automata occurs by using just a few bits of state information at every site to generate complex and interesting dynamical systems.

One of the simplest cellular automata is used to simulate the solution of the nonlinear 1-dimensional advection-diffusion equation, known as Burgers' equation, and written

\[ \rho_t + \rho \rho_x = \nu \rho_{xx}. \]

(Burgers' equation) evolves a sinusoidal initial condition into a shock front (or the reverse) and is often used to model flow in shallow channels. The speed of the shock formation is governed by the diffusion coefficient \( \nu \). The automaton uses a one-dimensional periodic grid of sites. Each site is represented by 2 bits, each either 0 or 1. One bit represents the presence or absence of a particle moving to the left, while the second bit represents the presence or absence of a particle moving to the right. The automaton evolves by advecting the particles to their respective adjacent sites and then colliding particles site-by-site according to a built-in rules table. Not all of the rules are deterministic, some are probabilistic. The outcome of a collision involving the probabilistic rules is dependent upon the value of a random bit.

The simulation thus requires the generation of a sequence of random bits, not a sequence of random numbers. It would be expected that the numerical average of randomly generated bits be equal to \( \frac{1}{2} \). This is due to the fact that roughly half of the bits will have the value 0 and roughly half will be 1. The collision rules of the lattice gas introduce a bias into the randomness however. That is, the random bits have a mean not equal to \( \frac{1}{2} \), but instead equal to a calculated value (slightly greater than \( \frac{1}{2} \)) based on the size of the problem. The value of a random number will be used to probabilistically flip a 0-bit so that the mean of the bit stream is the calculated value. The resulting bit stream will then be applied during the collision phase of the Burgers' equation automaton.

Random number, and random bit, generation is often the processing bottleneck of stochastic cellular automata and Monte Carlo simulation. For this particular lattice gas the UNIX profiling utility `prof`\(^1\) shows that more than \( \frac{1}{6} \) of the running time for the lattice gas is spent generating random numbers, so decreasing this bottleneck must occur if

---

\(^1\) `prof` is a tool that samples the machine's program counter during execution and returns information
large problems are to be computed. An efficient technique to generate a stream of biased random bits in parallel, while meeting appropriate statistical criteria for the bits and random numbers from which they derive, needs to be developed in order to run larger, and more physically meaningful, problems on today's computers.

The end purpose of the thesis is to determine an efficient method for implementing a lattice gas which models Burgers' equation. Lattice gases in general involve simple bit-wise operations applied to many sites for many times steps. The lattice gas for Burgers' equation is stochastic and therefore requires the generation of biased random bits to determine the effect of particle collisions within the simulation. The collision and advection phases are each highly parallel. The implementation of a random number generator which exhibits good distributional and statistical properties, is efficient, and can be parallelized will be developed. Different types of random numbers generators will be looked at and their properties investigated. The random number generators will include standard generators such as multiplicative congruential and lagged Fibonacci, both often available as system functions to programmers, and shift register, which is easy to encode in hardware. A less well known, but effective generator based on a cellular automaton will be looked at. The generators will be subjected to statistical tests such as simple moments and more complex $\chi^2$ tests. These tests measure the long term statistical properties of the generated sequence of random numbers as well as their near-term statistical properties. The period, or length before repetition, will be examined. The period of generators becomes more important as problem size grows, and is sometimes a limit on the effectiveness of a generator.

Once a generator is found which can produce the necessary number of random values and meet the statistical test of randomness the random numbers will be used to derive a stream of random bits. There is a stringent $\chi^2$ test which will be used to ensure that the bits are individually random. It will be these random bit streams that will be utilized to implement the lattice gas on two massively parallel machines: the BBN TC2000 and the CRAY C90.

Both the TC2000 and the C90 are considered MIMD—multiple instruction, multiple data—computers according to the classification (of computer architectures) first suggested on the number of clock cycles used by and the amount of execution time spent in each procedure. There are also utilities which can monitor the execution of the program at the level of small blocks of code. Such data can be used to find computational bottlenecks or hotspots in the code.
by Flynn [HP90] and now widely accepted. Each machine is capable of allowing the individual processors to execute separate streams of instructions with different streams of data. It's also possible to run either machine with the individual processors executing the same instructions using different pieces of data. The BBN is a distributed-memory switched system [HJ88]. The memory is distributed as local memory to the individual processors, but is viewed as a single, global memory. The processors in turn are connected to each other using a multistage packet switch. At the programmer level, all data is seen to be shared by the processors. Since the data is distributed across the processors, memory access time may differ for different data. This is because some data is local to the processor needing the data, while other data resides on different processors and must be passed along the communication network. Communication between the processors occurs using either a split-join model, where communication is controlled by the system, or a message-passing model, where communication is controlled by the user. The split-join concept allows users to assume the computer is a virtual multi-processor. The processors form a team that executes the code from start to finish. Message Passing assumes that each processor has its own local memory and all sharing of data occurs through messages passed between the processors. The message passing model requires that all processes execute the user program from start to finish.

The CRAY is a true shared memory machine. There is one common memory, and all processors maintain an equal access time to the memory. Processor communication occurs under user control using Multitasking. Multitasking lets both the user and the compiler control the parallelization. The compiler can parallelize loops automatically using autotasking. The user can do additional fine-grained parallelism using microtasking. Microtasking allows the user to specify which sections of the code can be done in parallel [MST93]. The code can also be parallelized using macrotasking, with which the code is partitioned into individual tasks that can be executed concurrently. Multitasking as a whole can be thought of as subroutine-level parallelism [DDSL93].

The lattice gas is fairly easy to implement. The straightforward implementation is to generate one random number at each lattice site, which is immediately used, in two or more steps, to create the necessary biased random bit for that site. The thesis will try to find a more efficient solution whereby one generated random number will be used to
create thirty-two biased random bits. The random number generator must not be difficult or costly to implement. It must meet statistical requirements over the entire sequence and over near-term subsequences. It must also have each of its individual bits be statistically random. In addition, the generator needs to have a period sufficiently long to allow large simulations to be run. Using the bits from a random number as random bits and biasing them as a group (one word’s worth at a time) requires extra work at each step. It is hoped that the extra computation required to access each of the bits in a random number will be more than offset by the decrease in time necessary to generate the random numbers. If this is indeed the case, then the computational bottleneck created by the need to generate a large stream of random numbers will be diminished.
Chapter 2

Random Number Generation

2.1 Introduction

Ideally, random numbers would be generated in a manner both independent of the application being computed and be physically stochastic. Martino and Morris [MM91] have implemented one such generator which generates random numbers based on the 2-dimensional position of detected photo-events using optical position-sensitive photon-counting detectors. Other methods might involve measuring small changes in the voltage or current across a set of input lines; or measuring the timing or direction of decay from a small radioactive source material. Monitoring the background radiation at the computer site might even be considered. Each of these methods, while truly random in the sense of unpredictability and nondeterminism, requires extraneous hardware to generate a single random number and is much too inefficient to generate the billions of random numbers needed in typical computations.

Since hardware generation of random numbers is impractical, software must be used to create the values needed. There are many algorithms for generating random numbers. As long as the algorithms are implemented with care, the values generated are acceptable statistically as random numbers. It has been noted that because an algorithm is a deterministic process, we cannot generate truly random numbers, but only pseudo-random numbers. However, if one were given a sequence of numbers, which meet some defined idea of randomness, and—without knowing the specific algorithm used—were unable to predict the next value in the sequence, then the sequence of pseudo-random numbers should be
CHAPTER 2. RANDOM NUMBER GENERATION

considered a sequence of random numbers.

A pseudo-random number generator will have certain desirable properties if it is to be considered acceptable [Jam90]: the generator needs to be *computationally efficient*—it should be implemented easily and execute quickly; *portable*—it should generate the exact same sequence on different machines, given the same seed; and *repeatable*—for testing, debugging, and the ability to restart a calculation at any point along the sequence. Sherif and Dear [SD93] add the two additional criteria *independence*—the generated numbers should not be correlated with each other; and *storage efficiency*—the generator should not require too much memory. The most important properties though, are a long *period* ($\tau$) and good *distribution* (also known as *uniformity*).

Being an algorithmic computation, a random number generator "cannot produce a non-repeating sequence without external interference" [ADG70]. The period is the length of the cycle that occurs when a number in the sequence is regenerated and the sequence begins to repeat. The cycle is also known as a Poincaré period [Bro84]. It has been suggested that in order to ensure good distribution, only $0.10\% - 1.00\%$ of the full period of a sequence be used [Deá90]. Good distribution corresponds to the intuitive notion of randomness—the random numbers should be spread evenly throughout the range of possible values and the knowledge of the most recently generated random number shouldn’t make the value of the next number to be generated more predictable. If the range of possible random numbers is subdivided into equal length partitions, each subrange should have approximately the same amount of numbers in it. This is commonly referred to as *equidistribution* [Nie87]. In addition, good distribution requires that successively generated random numbers be statistically independent (i.e. *uncorrelated*). These two notions of good distribution will be more rigorously defined later in the chapter by making use of probability theory’s $\chi^2$ statistic. Knuth[Knu81] gives a detailed and comprehensive theoretical analysis of periodicity and distribution (and random numbers in general).

2.2 Pseudo-Random Number Generators

Implementation of the lattice gas described in the following chapter is dependent upon the ability to generate random numbers. From Table 3.2 in the next chapter, the generators must have a period of least $1.1 \times 10^{12}$ to generate enough numbers for the case of $N = 16384$
lattice sites. To generate the stream of random values needed for the lattice gas, three simple generators were implemented. Specifically, the generators were multiplicative congruential, shift-register, and lagged Fibonacci. In addition, a cellular automaton was also used to generate random numbers. The generators were used to produce a sequence of random number uniformly distributed on the interval \([0, 1)\).

### 2.2.1 Multiplicative Congruential Generators

Multiplicative congruential generators are one of the oldest generators still commonly used. They were first proposed by Lehmer in 1951 [Leh51]. Standard library routines such as \texttt{rand} or \texttt{ranf} are usually congruential generators. Such a generator is denoted by \(MC[a,c,m]\), where

\[
\begin{align*}
    a & \quad \text{multiplier} \quad 0 < a < m \\
    c & \quad \text{additive constant} \quad 0 \leq c < m \\
    m & \quad \text{modulus}.
\end{align*}
\]

The sequence of random numbers generated is determined by

\[
x_{n+1} = (a \cdot x_n + c) \mod m, \quad n \geq 0,
\]

where,

\[
x_0 \quad \text{initial value} \quad 0 \leq x_0 < m.
\]

A congruential generator has a period less than or equal to the modulus. This implies \(m\) should be large, but if \(m\) is greater than the largest integer allowed in the machine, the efficiency of the congruential generator will be lost. Either the generator will have to be implemented with floating point values, whose operations are much slower than those for integers, or more than one word for each parameter will need to be used at each step in the computation, which also slows the implementation. The modulus is therefore chosen to be near the word size of the machine.

To ensure that the generator has full period—that is, all \(m\) possible values will occur in the generated sequence before any are repeated—or nearly full period, only certain combinations of \(a, c,\) and \(m\) will work (see Knuth [Knu81] for details). Some often used congruential generators are
Each of the generators was used to generate a stream of $10^6$ random numbers, which were checked for randomness on average. As can be seen from Table 2.1, all the multiplicative congruential generators exhibit good average statistical properties as defined in Section 2.3.1. Most platforms are 32-bit machines; the generators should have a modulus no larger than the size of one 32-bit integer, which eliminates the latter two generators.

<table>
<thead>
<tr>
<th>$MC[a,c,m]$</th>
<th>$\bar{x}$</th>
<th>$\int_0^1 x , dx$</th>
<th>$\int_0^1 \int_0^1 x , y , dx , dy$</th>
<th>$\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MC[65539,1,2^{32}]$</td>
<td>0.499829</td>
<td>0.353227</td>
<td>0.249895</td>
<td>$O(10^9)$</td>
</tr>
<tr>
<td>$MC[69069,1,2^{32} - 209]$</td>
<td>0.499977</td>
<td>0.333271</td>
<td>0.250086</td>
<td>$O(10^9)$</td>
</tr>
<tr>
<td>$MC[16807,0,2^{31} - 1]$</td>
<td>0.500270</td>
<td>0.334912</td>
<td>0.249847</td>
<td>$O(10^9)$</td>
</tr>
<tr>
<td>$MC[5^{15},7261067085,2^{35}]$</td>
<td>0.498092</td>
<td>0.332957</td>
<td>0.249093</td>
<td>$O(10^{10})$</td>
</tr>
<tr>
<td>$MC[2875A2E7B175_{16},0,2^{48}]$</td>
<td>0.499767</td>
<td>0.333096</td>
<td>0.249653</td>
<td>$O(10^{14})$</td>
</tr>
</tbody>
</table>

Table 2.1: Average properties of Multiplicative Congruential generators

Knuth [Knu81] has shown that the period of the 32-bit generator is $\approx 2^{32} \approx 4 \times 10^9$ and the period of the 31-bit generator is generator is $\approx 2^{31} \approx 2 \times 10^9$. This appears to be plenty of random numbers, but for the small lattice gas problem using 4096 sites, approximately $17 \times 10^9$ random numbers are needed. Since the BBN TC2000 and most workstations are limited to 32-bit words, congruential generators are not acceptable because their period is too short for use in the lattice gas automaton (or longer period generators will be inefficient to implement since they require two words of memory for the computation). It has been demonstrated [APG92] that multiplicative congruential generators are unsatisfactory not only due to having too short a period for parallel applications, but the sequence of generated numbers also exhibits regularities which make the sequence not sufficiently random.
2.2.2 Shift Register Generators

Shift register generators (also known as Tausworthe or Generalized Feedback Shift Register generators) are denoted by $\text{SR}[r,l,m]$ where

- $r$: number of bits to shift right (end off, zero fill) \hspace{1cm} r > 0$
- $l$: number of bits to shift left (end off, zero fill) \hspace{1cm} l > 0$
- $m$: number of bits \hspace{1cm} r, l < m.$

The sequence of random numbers generated is determined by

$$x_{n+1} = (I \otimes R')(I \otimes L') x_n, \; n \geq 0.$$  

Here, $\otimes$ is the bitwise XOR operation, $R'$ is a shift right $r$ bits, and $L'$ is a shift left $l$ bits. $x_{n+1}$ is computed in two steps. First, $x_n$ is XOR'd with its own left shift. Second, the result is then XOR'd with its right shift.

A shift register generator has a period less than or equal to $2^m$. This implies $m$ should be large, but if $m$ is greater than the word length of the machine, the efficiency of the generator is diminished. The generator would have to be implemented with more than one word needed for each parameter at each step of the computation, which would slow down the generator. The number of bits of the generator is usually chosen to be the word length of the machine.

To ensure that the generator has full period—that is, all $2^m$ possible values will occur in the generated sequence before any are repeated—or nearly full period, only certain values of $r$ and $l$, based on primitive polynomials, will work (see Knuth [Knu81] for details). Some often used shift register generators are

- $\text{SR}[3,28,31]$,
- $\text{SR}[6,25,31]$,
- $\text{SR}[13,18,31]$,
- $\text{SR}[15,17,32]$,
- $\text{SR}[7,9,16]$.

Each of the generators was used to generate a stream of $10^6$ random numbers, which were checked for randomness on average. As can be seen from Table 2.2, all the shift register generators exhibit good average statistical properties as defined in Section 2.3.1.
Most platforms are 32-bit machines; the generators should have a modulus the size of one 32-bit integer, eliminating the last generator due to the waste of a halfword for each parameter at every step of the computation.

\[ \text{SR}[r,l,m] \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3,28,31</td>
<td>0.499704</td>
<td>0.332437</td>
<td>0.249627</td>
<td>( O(10^9) )</td>
</tr>
<tr>
<td>6,25,31</td>
<td>0.497821</td>
<td>0.333207</td>
<td>0.249868</td>
<td>( O(10^9) )</td>
</tr>
<tr>
<td>13,18,31</td>
<td>0.498301</td>
<td>0.333195</td>
<td>0.249731</td>
<td>( O(10^9) )</td>
</tr>
<tr>
<td>15,17,32</td>
<td>0.500656</td>
<td>0.332897</td>
<td>0.249892</td>
<td>( O(10^9) )</td>
</tr>
<tr>
<td>7,9,16</td>
<td>0.499634</td>
<td>0.332874</td>
<td>0.249926</td>
<td>( O(10^4) )</td>
</tr>
</tbody>
</table>

Table 2.2: Average properties of Shift Register generators

Anderson [And90] has shown that the period of the three 31 bit generators have a period of \( \approx 2^{31} \approx 2 \times 10^9 \) and the 32-bit generator has a period of \( \approx 2^{32} \approx 4 \times 10^9 \). As with the multiplicative congruential generators above, the period of the the register generators is inadequate for use with the lattice gas automaton.

### 2.2.3 Lagged Fibonacci Generators

Lagged Fibonacci generators are becoming more popular, being the basis for many built-in \texttt{random} library routines. The name is loosely based on the idea of the Fibonacci series, defined as the recurrence

\[ x_{n+2} = x_{n+1} + x_n, (n \geq 0, x_1 = x_0 = 1). \]

Instead of using the two immediately preceding terms, lagged Fibonacci generators use two terms which can be much further back in the sequence and need not be consecutive. The distance to look back in the sequence is called the \textit{lag}. The generators are denoted by \texttt{LF}[r,s,m,op] where

- \( r \) \textit{lag} \( r > 0 \)
- \( s \) \textit{lag} \( r > s \)
- \( m \) \textit{base}
- \( op \) \textit{operation} \( + \text{mod } m, - \text{mod } m, * \text{mod } m, \text{ or } \odot \) if \( m \) is a power of 2.
The sequence of random numbers generated is determined by

\[ x_n = (x_{n-r}) op (x_{n-s}) , \quad n \geq r. \]

It should be noted that a sequence of the last \( r \) values needs to be maintained. In addition, the sequence must be initialized by using another random number generator to generate the first \( r \) values.

Unlike congruential or shift generators, the period of Fibonacci generators can be much greater than the base \( m \). This makes the generators well suited for applications that require many billions of random numbers even though a short sequence of previous random numbers must be carried along. To ensure that the generator has maximal period, only certain values of \( r \) and \( s \) (and appropriate operations), based on the theory of linear recursive sequences of integers, will work (see Marsaglia [Mar85] for details). Some often used lagged Fibonacci generators are

\[
\begin{align*}
\text{LF}[55,24,32,+], \\
\text{LF}[17,5,32,\pm], \\
\text{LF}[607,273,31,-], \\
\text{LF}[31,13,32,*], \\
\text{LF}[1279,418,32,\oplus].
\end{align*}
\]

Each of the generators was used to generate a stream of \( 10^6 \) random numbers, which were checked for randomness on average. As can be seen from Table 2.3, all the lagged Fibonacci generators exhibit good average statistical properties as defined in Section 2.3.1.

<table>
<thead>
<tr>
<th>LF([r,s,m,op])</th>
<th>( \bar{x} )</th>
<th>( \int_0^1 x , dx )</th>
<th>( \int_0^1 \int_0^1 x , y , dx , dy )</th>
<th>( \tau )</th>
</tr>
</thead>
<tbody>
<tr>
<td>55,24,32,+</td>
<td>0.500121</td>
<td>0.333455</td>
<td>0.250160</td>
<td>( O(10^{25}) )</td>
</tr>
<tr>
<td>17,5,32,\pm</td>
<td>0.500872</td>
<td>0.333476</td>
<td>0.250172</td>
<td>( O(10^{14}) )</td>
</tr>
<tr>
<td>607,273,31,-</td>
<td>0.499534</td>
<td>0.332789</td>
<td>0.249561</td>
<td>( O(10^{19}) )</td>
</tr>
<tr>
<td>31,13,32,*</td>
<td>0.500180</td>
<td>0.333512</td>
<td>0.250052</td>
<td>( O(10^{18}) )</td>
</tr>
<tr>
<td>1279,418,32,\oplus</td>
<td>0.499089</td>
<td>0.333364</td>
<td>0.250055</td>
<td>( O(10^{39}) )</td>
</tr>
</tbody>
</table>

Table 2.3: Average properties of Lagged Fibonacci generators
Marsaglia [Mar85] has shown that the period of the LF[55,24,32,+] generator is \((2^{55} - 1)(2^{31}) \approx 64 \times 10^{24}\), which is long enough for largest problems currently being run even taking into account Deak's [Dea90] 0.10% rule of thumb. The generators with lags of 607 and 1279 offer extremely long periods but are not implemented because the 55-lag generator has a more than adequate period. Additionally, the 607- and 1279-lag generators require 11 and 23 times as many values respectively be maintained as the for 55-lag generator. The 17-lag generator has a period that is too small for the lattice gas automaton. The 31-lag generator has a period acceptable for the automaton, but the period is too short if larger problems are to be run. The lagged Fibonacci generator LF[55,24,32,+] was implemented to determine if it could be used to generate the random bits needed by the lattice gas.

2.2.4 Cellular Automata Generators

Cellular automata are a class of mathematical systems which are characterized by discreteness in space, time, and state value. Spatial discretization implies that there is a lattice or grid on which sites reside. In addition, there is the idea of a neighborhood, i.e. those sites immediately adjacent to a particular site. Temporal discretization means the evolution of the cellular automaton proceeds in time steps. Discretization of the state values, or the dependent variable, allows the values to be represented by a number of bits, which theoretically could be infinite, at each site. The power of cellular automata occurs by using just a few bits for state information at every site to generate complex and interesting dynamical systems.

One of the simplest forms of cellular automata uses a one-dimensional periodic grid of sites, where each site is represented by a single bit, either 0 or 1. Each site is randomly initialized at time\(_0\). At time\(_1\), and all subsequent times, a site is updated according to the binary value of its neighborhood—where the neighborhood of a site consists of itself and its immediate left and right neighbors. The updated, or new, value of a site is found by applying a rule to the binary value of the neighborhood. All sites are updated in unison at each time step. The rules for this class of automata are derived by converting a rule number's decimal value to binary and creating a lookup table to be used for updating. This class of cellular automata is known as \(k = 2, r = 1\) automata [Wol84]. The name \(k = 2, r = 1\) derives from updating a cell's value based on the values 1 generation previously of its 2 nearest
neighbors, along with itself. The popular computer recreation known as Conway’s LIFE is a 2-dimensional $k = 4, r = 1$ cellular automaton.

The $k = 2$ neighborhood of a site $i$ consists of three sites: left neighbor $(i - 1)$, itself $(i)$, and right neighbor $(i + 1)$. Each site is denoted by a bit, it can only possess the values 0 and 1. The neighborhood can then be represented as a 3-bit binary value, which can take on any of $2^3$, or 8, possible sequences or values. The number of possible values is also the length of the rule lookup table. The indices of the lookup table are the neighborhood values ordered by increasing binary value (000, 001, 010, ..., 111). The output of the $j$th entry in the table is the $j$th least most significant bit of the binary representation of the rule number. The table has 8 entries, which allows for 8-bit rule numbers, which can have values from 0 up to $2^8$ (or 255).

Suppose 30 is chosen as the rule number. The 8-bit binary value of 30 is 00011110. This value goes into the lookup table with the least most significant, or rightmost, bit in index 0. The table is set as follows:

<table>
<thead>
<tr>
<th>000</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.4: Lookup table for rule 30, $k = 2, r = 1$ automaton

The left column, or index, entries are the 8 possible binary sequences from the 3-term neighborhood. The second column is the update value for the center site of the neighborhood. If the respective bit values for the neighborhood of $i$ are, for example, 010 at time $t$, the new value at time $t+1$ for site $i$ is 1. If the values are 101, then the updated value is 0.

Wolfram [Wol86] defines two $k = 2, r = 1$ automata which exhibit acceptable random number properties. The two rules are 30 and 60. The automata can be made any length, that is, the automata can be made with an arbitrary number of bits. Different automata lengths yield random number generators with different statistical and cyclic (i.e. periodic)
properties. For both the rule30 automaton and the rule60 automaton, the longest cycle is created using an automaton with 53 bits. For each of the two automata, a 32-bit random number was generated by probing the middle 32 bits of the automaton at each iteration. Each of the generators was used to generate a stream of $10^6$ random numbers, which were checked for randomness on average. As can be seen from Table 2.5, the cellular automaton rule30 generator and the rule60 generator both exhibit good average statistical properties as defined in Section 2.3.1.

<table>
<thead>
<tr>
<th>rule</th>
<th>$\bar{x}$</th>
<th>$\int_0^1 x , dx$</th>
<th>$\int_0^1 \int_0^1 x , y , dz , dy$</th>
<th>$\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.499896</td>
<td>0.333875</td>
<td>0.250137</td>
<td>$O(10^{11})$</td>
</tr>
<tr>
<td>60</td>
<td>0.499693</td>
<td>0.334472</td>
<td>0.249913</td>
<td>$O(10^{10})$</td>
</tr>
</tbody>
</table>

Table 2.5: Average properties of Cellular Automaton generators

It may be noted that the period for the rule30 generator is just adequate enough for the lattice gas application. Although the rule30 generator exhibits both good distribution and long period, Figure 2.1 shows that the generator is roughly three times slower than LF[55,24,32,+]· Since the aim was to implement random number generation efficiently, the cellular automaton generator was not implemented for the lattice gas, only the lagged Fibonacci generator was implemented.

2.3 Statistical Measurements of Generators

A difficulty associated with random number generation is determining if the sequence of numbers generated is indeed random. Graphical representation of the generated numbers over a full period, or most of the cycle, presents a visual degree of a generator's randomness [And90]. Each point in the graph is created by generating a random number and letting the value be the $y$-coordinate, while the previous random number is the $x$-coordinate. A good generator will create a lattice structure where each point has adjacent points equidistant along the lattice axes (see Figure 2.2). A generator that is poor will still have a lattice structure, but the points along one axis will be much closer than those along the other axis (see Figure 2.3). Random number generators with long periods don't lend themselves to
graphical inspection because the plots become too dense to distinguish whether the lattice structure consists of equally spaced points. Anderson ([And90]) has also noted that while all generators produce plots with some structure, only multiplicative congruential generators always produce plots with a lattice structure. Instead, arithmetic measures must be used to determine whether a generator produces a sequence of truly random numbers.

Often, measuring the average properties of a sequence provide insight into the degree of randomness. Average properties can be useful as a first step in determining if a generator is producing random numbers, but they cannot be used to guarantee that a sequence is random. Adjacency tests exist to test whether sequential, or any pair of, numbers are related. A set of adjacency tests must be used to be confident that a generated sequence is statistically random. The adjacency tests involve confidence tests based on $\chi^2$ statistics. These statistical values can yield insight into the overall properties of a long sequence of random numbers. Other tests are necessary to ensure that the generators return their values in a near-term random fashion also. These tests are also based on $\chi^2$ statistics, but on short subsequences of the generated numbers. Additionally, the overlapping $m$-tuple test, described later in the chapter, is used measure the randomness of the individual bits.
in each random number.

2.3.1 Average Properties

The average properties measured are the among the simplest. They are more commonly known as the sample moments of the sequence. The mean of each sequence was measured, which should equal $\frac{1}{2}$, since the random numbers are assumed to be uniformly distributed on the interval $[0,1)$. This is the first sample moment. Two correlations were also measured.

The first is the mean of the square of the numbers. It is expected that this value should equal $\frac{1}{3}$, since generating uniform random numbers and squaring their values is equivalent to the integral $\int_0^1 x \, dx$. The second is a serial correlation. This value is the mean of the product of successive random numbers. It is expected that this value should equal $\frac{1}{4}$, since generating
uniform random numbers in succession is equivalent to the integral $\int_0^1 \int_0^1 xy \, dx \, dy$. These are the second and third sample moments respectively. Higher moments may also be generated. All the random number generators explored earlier showed excellent agreement with the expected values of the average properties.

2.3.2 Ordered and Unordered $\chi^2$ Tests

Simple $\chi^2$ tests are used to ensure that the generated numbers are uniformly distributed. They statistically validate whether the sequence of random numbers fills up subintervals of the range in an even manner, i.e., equidistribution. The entire interval $[0,1)$ is divided into a number of subintervals, here chosen to be 10. As each random number is generated it is put into the bin for its appropriate subinterval and a counter in the subinterval is incremented. Once the proper amount of random numbers has been generated (equal to
that necessary for a particular lattice gas implementation), the number of random numbers in each subinterval is then compared with the expected count in that subinterval. The resulting $\chi^2$ statistic is used to determine whether the actual counts of random numbers in each subinterval is consistent with the statistically expected counts. This is often referred to as the \textit{chi-square goodness-of-fit test} [OD75]. Since the random numbers are assumed to be uniformly distributed on $[0, 1)$, each subinterval would be expected to contain equal amounts of generated numbers. Specifically, each subinterval should have 10% of the generated numbers.

Three similar tests were performed. The first was the \textit{unordered $\chi^2$ test}, written as $\chi_1^2$. This test simply measures the expected count versus the actual count for each of the subintervals. The formula is

$$\chi_1^2 = \frac{k}{n} \sum_{i=1}^{k} \left( f_i - \frac{n}{k} \right)^2$$

where

- $n$ number of random numbers generated
- $k$ number of subintervals
- $f_i$ observed count in subinterval $i$.

For large $n$, $\chi_1^2$ follows a $\chi^2$ distribution with $k - 1$ degrees of freedom [HD62]. With 10 subintervals, there are 9 degrees of freedom. This is a frequency test in which the sample moments don’t depend on the actual order the random numbers are generated.

The other two tests use the term computed above and an additional term. The new term is defined by the formula

$$\chi_2^2 = \frac{k^2}{n} \sum_{i=1}^{k} \sum_{j=1}^{k} \left( f_{ij} - \frac{n}{k^2} \right)^2$$

where

- $n$ number of random numbers generated
- $k$ number of subintervals
- $f_{ij}$ observed count of a number in subinterval $i$ followed by a number in subinterval $j$.

The tests are \textit{ordered $\chi^2$ tests}, written as

$$\chi_2^2 - \chi_1^2$$
and

\[ \chi_2^2 - 2\chi_1^2. \]

These two tests measure how often a random number in subinterval \( j \) follows one generated in subinterval \( i \). For large \( n \), \( \chi_2^2 - \chi_1^2 \) follows a \( \chi^2 \) distribution with \( k(k - 1) \) degrees of freedom and \( \chi_2^2 - 2\chi_1^2 \) follows a \( \chi^2 \) distribution with \( (k - 1)^2 \) degrees of freedom [HD62]. The two tests have 90 and 81 degrees of freedom respectively. These are serial tests in which the sample moments depend on the specific order in which the random numbers are generated.

Each of the three \( \chi^2 \) tests defined above yield a value which is then used to compute a confidence measure of whether the actual distribution of random numbers is uniform. The expected values in each term are theoretical probability distributions. The value returned from a test is compared with the value in a \( \chi^2 \) distribution table. The proper index into the table is the degrees of freedom. The columns of the lookup table are the numerical value of the theoretical \( \chi^2 \) distribution. The columns are arranged by a confidence percentage (commonly referred to as the level of significance \( \alpha \)). If the computed test value is greater than the corresponding value in the table, then the actual distribution is assumed to be random with a certainty of at least the percentage of the column.

To understand how the \( \chi^2 \) statistic is used, an illustration using 1048576 random numbers generated by LF[55,24,32,+] and the unordered \( \chi^2 \) test (\( \chi_1^2 \)) will help. Table 2.6 lists the subintervals and the number of random numbers generated in each subinterval. These are the actual values. Since the generator is assumed to generate numbers uniformly on \([0,1)\), each subinterval should be expected to contain \( \frac{1}{10} \)th of the generated numbers, or 104857.6.

The resulting value for \( \chi_1^2 \) is 6.18 with \( k = 10 \) subintervals and 9 degrees of freedom (df). \( \chi_1^2 \) can now be used to determine if the distribution of random numbers is statistically random. The significance value is the confidence \((0 < \alpha < 1)\) with which the data can be said to be not due to chance. The \( \chi^2 \) distribution table has the following row entry for 9 degrees of freedom [RW90]:

<table>
<thead>
<tr>
<th>df</th>
<th>( \alpha = 0.01 )</th>
<th>( \alpha = 0.05 )</th>
<th>( \alpha = 0.25 )</th>
<th>( \alpha = 0.50 )</th>
<th>( \alpha = 0.75 )</th>
<th>( \alpha = 0.95 )</th>
<th>( \alpha = 0.99 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2.088</td>
<td>3.325</td>
<td>5.899</td>
<td>8.343</td>
<td>11.39</td>
<td>16.92</td>
<td>21.67</td>
</tr>
</tbody>
</table>
Knuth [Knu81] suggests that $\chi^2$ values above $\alpha = 0.99$ be rejected as too random and values below $\alpha = 0.01$ be rejected as not random enough. Values ranging between $0.90 < \alpha < 0.99$ and $0.01 < \alpha < 0.10$ should be considered suspect or almost suspect. Values between $0.25 < \alpha < 0.75$ are acceptable as they are neither significantly low nor significantly high. The computed value of $\chi^2 = 6.18$ falls between $\alpha = 0.25$ and $\alpha = 0.50$ and is therefore deemed acceptable.

The computed value for $\chi^2$ is based on the ordered frequency data from Table 2.7. There are ten subintervals, and each number can be put into any of ten previous subintervals. This means that the expected value for each ordered subinterval is $\frac{1}{100}$ of the generated random numbers, or 10485.76.

As can be seen from Table 2.8, the LF[55,24,32,+] generator passes the unordered and ordered tests. The significance values used to reject the null hypothesis are exact for the unordered test and estimated for the two ordered tests [Knu81]. In order to be declared acceptable, the $\chi^2$ values had to be between the $\alpha = 0.25$ and $\alpha = 0.75$ levels of significance. Other generators aren't listed in the table because they had previously been rejected for use with the lattice gas application due to too short a period or too inefficient an implementation.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0.0000 - 0.0999</td>
<td>104744</td>
</tr>
<tr>
<td>2 0.1000 - 0.1999</td>
<td>105057</td>
</tr>
<tr>
<td>3 0.2000 - 0.2999</td>
<td>104912</td>
</tr>
<tr>
<td>4 0.3000 - 0.3999</td>
<td>104956</td>
</tr>
<tr>
<td>5 0.4000 - 0.4999</td>
<td>104508</td>
</tr>
<tr>
<td>6 0.5000 - 0.5999</td>
<td>104762</td>
</tr>
<tr>
<td>7 0.6000 - 0.6999</td>
<td>104476</td>
</tr>
<tr>
<td>8 0.7000 - 0.7999</td>
<td>105410</td>
</tr>
<tr>
<td>9 0.8000 - 0.8999</td>
<td>104866</td>
</tr>
<tr>
<td>10 0.9000 - 0.9999</td>
<td>104885</td>
</tr>
</tbody>
</table>

Table 2.6: Table of unordered subinterval frequencies for LF[55,24,32,+]
CHAPTER 2. RANDOM NUMBER GENERATION

<table>
<thead>
<tr>
<th>INTERVAL</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10518</td>
<td>10483</td>
<td>10547</td>
<td>10576</td>
<td>10495</td>
<td>10494</td>
<td>10382</td>
<td>10364</td>
<td>10408</td>
<td>10477</td>
</tr>
<tr>
<td>2</td>
<td>10485</td>
<td>10644</td>
<td>10448</td>
<td>10519</td>
<td>10330</td>
<td>10472</td>
<td>10605</td>
<td>10554</td>
<td>10589</td>
<td>10411</td>
</tr>
<tr>
<td>3</td>
<td>10535</td>
<td>10328</td>
<td>10449</td>
<td>10629</td>
<td>10431</td>
<td>10427</td>
<td>10341</td>
<td>10608</td>
<td>10598</td>
<td>10565</td>
</tr>
<tr>
<td>4</td>
<td>10457</td>
<td>10582</td>
<td>10394</td>
<td>10447</td>
<td>10502</td>
<td>10585</td>
<td>10391</td>
<td>10706</td>
<td>10334</td>
<td>10558</td>
</tr>
<tr>
<td>5</td>
<td>10311</td>
<td>10421</td>
<td>10465</td>
<td>10378</td>
<td>10445</td>
<td>10703</td>
<td>10402</td>
<td>10432</td>
<td>10529</td>
<td>10422</td>
</tr>
<tr>
<td>6</td>
<td>10429</td>
<td>10645</td>
<td>10410</td>
<td>10435</td>
<td>10361</td>
<td>10521</td>
<td>10620</td>
<td>10544</td>
<td>10341</td>
<td>10465</td>
</tr>
<tr>
<td>7</td>
<td>10498</td>
<td>10489</td>
<td>10460</td>
<td>10437</td>
<td>10341</td>
<td>10278</td>
<td>10368</td>
<td>10496</td>
<td>10586</td>
<td>10543</td>
</tr>
<tr>
<td>8</td>
<td>10460</td>
<td>10468</td>
<td>10545</td>
<td>10495</td>
<td>10646</td>
<td>10515</td>
<td>10452</td>
<td>10647</td>
<td>10606</td>
<td>10576</td>
</tr>
<tr>
<td>9</td>
<td>10541</td>
<td>10643</td>
<td>10570</td>
<td>10465</td>
<td>10411</td>
<td>10510</td>
<td>10512</td>
<td>10526</td>
<td>10338</td>
<td>10350</td>
</tr>
<tr>
<td>10</td>
<td>10519</td>
<td>10373</td>
<td>10624</td>
<td>10575</td>
<td>10546</td>
<td>10257</td>
<td>10403</td>
<td>10533</td>
<td>10537</td>
<td>10518</td>
</tr>
</tbody>
</table>

Table 2.7: Table of ordered subinterval frequencies for LF[55,24,32,+] random number generator

<table>
<thead>
<tr>
<th>$X_1^2$</th>
<th>$X_2^2$</th>
<th>$X_1^2 - X_2^2$</th>
<th>$X_2^2 - 2X_1^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.18</td>
<td>89.33</td>
<td>83.15</td>
<td>76.97</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>---</td>
<td>ACCEPT</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

Table 2.8: $\chi^2$ tests of LF[55,24,32,+] random number generator

2.3.3 Distributional $\chi^2$ Tests

The $\chi^2$ tests described above help determine whether an entire sequence, or long subsequence, of random numbers is generated randomly over the entire sequence. It is often important to determine that the generator has random properties for very short subsequences also. Distributional tests can be used to look at these short-term properties. Such tests are considered to be runs tests on the sequence of random numbers. They are used to determine if short subsequences are random in a statistical sense. A few such tests are described below.

The frequency test is similar to the unordered $\chi^2$ test, but where many subsets of shorter subsequences of consecutive numbers are compared with the theoretical $\chi^2$ distributions. The lattice gas application lends itself well to the frequency test. The subsequences need only be as long the number of random numbers needed during each iteration of the simulation—$N$. The number of subsets is just the number of iterations the automaton is run through. As with the entire sequence of random numbers, the values generated within the subsequence are put into one of ten subintervals, and a count is kept of how many random
numbers are in each subinterval. The observed counts are then used to compute the value for $\chi^2$. The expected value for each subinterval is $\frac{1}{15}$th of the length of the subsequence.

The **poker test** uses the random numbers to compute 10-card "poker hands" from consecutive values. Numbers below 0.5 are called "red" cards, while those above 0.5 are called "black" cards. The poker test also requires many subsets of many hands; the lattice gas problem can set the length of the subsequences and the number of subsets. The resulting red-black hands are then compared with the theoretical $\chi^2$ distribution arising from $i$ red cards and $10 - i$ black cards $((\binom{10}{i}) \times 2^{-10})$ [ADG70]). These combinatoric values are the expected values.

The **runs above (runs below)** test too requires many subsets of shorter subsequences of random numbers. Sequences of consecutive numbers greater (less) than the median (0.5) preceded and followed by a number less (greater) than the median are known as runs above (below). Lengths of runs $k = 1, 2, \ldots, 9, \geq 10$ were tracked. The observed count of runs (above and/or below) of length $k$ is compared with the theoretical $\chi^2$ distribution resulting from the expected number of runs of length $k$ ($\approx n \cdot 2^{-k-1}$ [ADG70]). As with the frequency test, the lattice gas problem can set the length of the subsequences and the number of subsets.

The last test used is a **two-term serial correlation coefficient**, sometimes referred to as an **autocorrelation coefficient**. This measures how strongly two adjacent numbers in the generated sequence are related. When looking at pairs of successive numbers in the sequence (i.e. 1st and 2nd, 2nd and 3rd, 3rd and 4th, 4th and 5th, ...), the test computes **Pearson’s $r$ correlation coefficient**. This is defined as

$$r = \frac{\sum_{i=1}^{n-1} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\left[\sum_{i=1}^{n-1} (x_i - \bar{x})^2 \right] \left[\sum_{i=1}^{n-1} (y_i - \bar{y})^2 \right]}}$$

where

- $n$ number of random numbers
- $x_i$ sequence of random numbers
- $y_i$ sequence of successive random numbers (i.e. $y_i = x_{i+1}$)
- $\bar{x}$ mean of $x_i$
If the sequence of generated numbers is random in the sense that consecutive numbers are statistically independent, then \( r \approx 0.0 \). A perfectly correlated sequence—generating random numbers by incrementing by 1 for example—yields \( r = 1.0 \). Similarly, an ant-correlated sequence—decrementing by 1 to generate the next number—produces \( r = -1.0 \). A perfectly uncorrelated sequence yields \( r = 0.0 \). This test can be expanded to look at any sequence of pairs of numbers (e.g. 1st and 5th, 2nd and 6th, 3rd and 7th, 4th and 8th, 5th and 9th, 6th and 10th, ...), not just the adjacent ones [ADG70]. The correlation coefficient was computed not only for the entire sequence, but also for subsequences. The subsequences were the length of random numbers needed for each iteration of the lattice gas automaton, i.e. \( N \). This way, the maximum and minimum correlation coefficients could be computed (over the subsequences), as well as a variance for the subsequences.

As can be seen from Table 2.9, the LF[55,24,32,+] generator passes the distributional tests. The values in the table are from a run using subsequences of length 512. This allowed the number of subsets to be 512 also (see Table 3.2 for details). The significance values used to reject the null hypothesis are exact for the unordered test and estimated for the two ordered tests [Knu81]. In order to be declared acceptable, the \( \chi^2 \) values had to be between the \( \alpha = 0.25 \) and \( \alpha = 0.75 \) levels of significance. Other generators aren't listed in the table because they previously been rejected for use with the lattice gas application due to too short a period or too inefficient an implementation.

<table>
<thead>
<tr>
<th></th>
<th>frequency</th>
<th>poker</th>
<th>( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimum ( \chi^2 )</td>
<td>6.02</td>
<td>ACCEPT</td>
<td>6.87</td>
</tr>
<tr>
<td>average ( \chi^2 )</td>
<td>8.62</td>
<td>ACCEPT</td>
<td>9.15</td>
</tr>
<tr>
<td>maximum ( \chi^2 )</td>
<td>10.36</td>
<td>ACCEPT</td>
<td>11.24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>runs above</th>
<th>runs below</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimum ( \chi^2 )</td>
<td>5.98</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>average ( \chi^2 )</td>
<td>8.41</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>maximum ( \chi^2 )</td>
<td>10.75</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

Table 2.9: Distributional tests of LF[55,24,32,+] random number generator
2.4 Random Bit Generation and Testing

If the values generated by a random number generator are immediately used to compute a binary value (e.g. is \( \tau \geq 0.5 \)) then the computational effort used to generate an entire random number has essentially been wasted. It would be much more efficient to generate random bits. While random bit generation codes exist [PFTV92], they require nearly as much computational effort as for generating random numbers. In fact, the 53-bit cellular automaton generator described in Section 2.2.4 can be thought of as a random bit generator that in essence wastes 52 bits of information. This is due to having to update all 53 bits in the automaton, but only using 1 of those updated bits of information. The goal is thus to find a random number generator with acceptable properties over the entire sequence and use all the bits from each generated value as a sequence of random bits.

2.4.1 A Random Bit Generator

The previous sections have made it clear that random bits can be generated by using the individual bits in an acceptable random number generator, such as LF[55,24,32,+] or probing particular bits from a larger random number generator. This is not to suggest that random bits cannot be generated by themselves. In fact, there is a known random bit generator routine. It is based on primitive polynomials modulo 2 [PFTV92]. These are special polynomials whose coefficients are either 0 or 1. An example is

\[ z^{30} + z^6 + z^4 + z^1 + z^0, \]

which can be written in an abbreviated form by notating the nonzero powers of \( z \), i.e.

\[(30, 6, 4, 1, 0).\]

Press et al. note that "[e]very primitive polynomial modulo 2 of order \( n \) defines a recurrence relation for obtaining a random bit from the \( n \) preceding ones." In the above example, \( n = 30 \).

To generate the first random bit, a nonzero seed is generated (using the wall clock time is typical). The value of the random bit is the value of \( n \)th bit of the seed. The seed is then updated using the recurrence relation as bit-wise operations. If the \( n \)th bit is 0, the seed is left-shifted once. If the \( n \)th bit is 1, the seed is XOR'd with a mask whose bits
are the coefficients of the corresponding polynomial (except the \( n \)th bit, which is set to 0). This temporary value is left-shifted once and a 1 is placed into the rightmost bit. This value becomes the new seed and the process is repeated.

The recurrence relation, for any nonzero \( n \)-bit seed, produces a sequence of random bits whose period is of length \( 2^n - 1 \). That means the example above can generate \( 2^{30} - 1 \approx 10^9 \) random bits before repeating. There is a primitive polynomial modulo 2 for any \( n \), which means that the period of generated random bits can be made as long as necessary. For a typical 32-bit word recurrence (as in the example above), the random bit generator is almost four times slower than generating the same amount of bits from the lagged Fibonacci generator (see Figure 2.4). A random bit generator with a period long enough for use in the lattice gas application would require a primitive polynomial modulo 2 with an order of at least 40 \( (2^{40} \approx 10^{12}) \). Implementing a generator with \( n \) larger than 31 requires two words of 32-bit memory. Performing the random bit generation then becomes even slower when compared with random bit generation using the lagged Fibonacci generator.

![Graph](image.png)

Figure 2.4: LF\( [55,24,32,+] \) vs \( (30,6,4,1,0) \) random bit generation times
2.4.2  $m$-tuple Test

As mentioned above, the mean of the resulting random bits is a first step in ensuring that the bits are indeed random. The $\chi^2$ and distributional tests used for random numbers aren't meaningful with bit data, so other tests must be incorporated. Marsaglia[Mar85] recently proposed a useful test: the $m$-tuple test. The $m$-tuple test is a runs test performed on the string of bits returned from the random number generators, not on the numbers themselves.

The test works by generating a sequence of random numbers, then creating a new sequence of random numbers by using a succession of adjacent bits in consecutive numbers. This sequence is then subjected to the ordered $\chi^2$ test. Consider the sequence of integers (not normalized floating point values) $\{x_i\}$ from a random number generator. The random numbers can be written:

$$
x_1 = b_{1,1}b_{1,2}b_{1,3} \cdots b_{1,w}
$$
$$
x_2 = b_{2,1}b_{2,2}b_{2,3} \cdots b_{2,w}
$$
$$
\vdots
$$
$$
x_n = b_{n,1}b_{n,2}b_{n,3} \cdots b_{n,w},
$$

where $w$ is the generator word length. Marsaglia notes that by taking $x_{n+1} = x_1$ in the above equation and extending the sequence into a circular list, the distributional theory becomes more tractable.

The $m$-tuple test is performed on bits 1 to $l$ by first creating a new sequence of numbers $\{y_i\}$ made from the leading $l$ bits of each number in $\{x_i\}$:

$$
y_1 = b_{1,1}b_{1,2}b_{1,3} \cdots b_{1,l}
$$
$$
y_2 = b_{2,1}b_{2,2}b_{2,3} \cdots b_{2,l}
$$
$$
\vdots
$$
$$
y_n = b_{n,1}b_{n,2}b_{n,3} \cdots b_{n,l}.
$$

Using the sequence of values derived above, tuples of length $m$ are derived consecutively.
CHAPTER 2. RANDOM NUMBER GENERATION

That is, the $m$-tuples:

$$(y_1, y_2, y_3, \cdots, y_{m-1}, y_m)$$

$$(y_2, y_3, y_4, \cdots, y_m, y_{m+1})$$

$$\vdots$$

$$(y_{n-m+1}, y_{n-m+2}, y_{n-m+3}, \cdots, y_{n-1}, y_n)$$

$$(y_{n-m+2}, y_{n-m+3}, y_{n-m+4}, \cdots, y_n, y_1)$$

$$\vdots$$

$$(y_{n-1}, y_n, y_1, \cdots, y_{m-3}, y_{m-2})$$

$$(y_{n}, y_1, y_2, \cdots, y_{m-2}, y_{m-3}).$$

are generated one after the other. A count is kept of each occurrence of each of the $2^{m_l}$ possibles values of the $m$-tuples. The distribution of counts for each $m$-tuple value, $B_{m_l,i}$ is then compared with the theoretical $\chi^2$ value arising from the summation [Alt88]:

$$\chi^2 = \frac{2^{m_l}}{n} \sum_{i=1}^{2^{m_l}} \left( B_{m_l,i} - \frac{n}{2^{m_l}} \right)^2 - \frac{2^{(m-1)}l}{n} \sum_{i=1}^{2^{(m-1)}l} \left( B_{(m-1)l,i} - \frac{n}{2^{(m-1)}l} \right)^2. \quad (2.1)$$

The $m$-tuple test was run on all $w = 32$ bits, with $m = l = 3$. The bits were tested in the circular sequence:

1, 2, 3
2, 3, 4
\vdots
31, 32, 1
32, 1, 2.

The generator created a sequence of $\mathcal{O}(10^{12})$ random numbers. Each random number was used to create thirtytwo 9-tuples (each consists of 3 three-bit terms). The tuples were placed into one of the 512 ($2^{3 \times 3}$) bins. The observed counts were compared to the expected, equally distributed counts, using Equation 2.1. All thirtytwo bits were determined to be individually random—neither too random nor not random enough. It is interesting to note that while the lagged Fibonacci LF[55,24,32,+] generator was chosen as the random number generator to implement based on its long period, many of the shorter period multiplicative congruential and shift register generators would not have passed the $m$-tuple test. Altman has shown [Alt88] that multiplicative congruential generators exhibit "periodicity in the lower order
bits...and overrepresentation of certain short byte strings”. In fact, the lowest-order bit of a 32-bit multiplicative congruential generator will alternate between 0 and 1. Altman also showed that 32-bit shift register generators have high-order, central, and low-order bits which are not random. For 31-bit generators, Anderson [And90] showed multiplicative congruential generators have all 31 bits usable as random bits, as do shift register and lagged Fibonacci generators ([Alt88]).

2.4.3 Biasing the Bits

The mean of a sequence of random bits, assuming a uniform distribution, should be $\frac{1}{2}$. However, we wish to generate a sequence whose mean is determined by the run-time parameters of the lattice gas problem. That is, the bias in the lattice gas is determined by the lattice spacing and the spatial averaging of the problem size. It turns out that the bias is equal to the desired mean of the random bits. The question is then how to generate the bits so that the mean is the same as the bias? The common method would be to look at just enough consecutive bits and generate a biased bit based on the probability of finding that exact bit sequence.

As a specific example, assume each bit should have a mean of $\frac{3}{5}$. Look at each bit in turn, using the notation $\text{bit}_i$ for a bit as it is checked. If $\text{bit}_i$ is 1—which has a probability of $\frac{1}{2}$—move on to the next bit. If $\text{bit}_i$ is 0—which also has a probability of $\frac{1}{2}$—the two bit sequence $\text{bit}_i \text{bit}_{i+1}$ (where $\text{bit}_{i+1}$ is the right neighboring bit) is examined. As the sequence consists of two bits there are four bit patterns possible. If the sequence 00 occurs—which has a probability of $\frac{1}{4}$—then $\text{bit}_i$ is flipped from 0 to 1. If the sequence 01 occurs—which also has a probability of $\frac{1}{4}$—$\text{bit}_i$ is not flipped. Remember that the patterns 10 and 11 won’t be examined because $\text{bit}_i$ is 1 in each. Using the bitwise and operation, $\text{bit}_i \land \text{bit}_{i+1}$, it is computationally easy to check if the sequence is 00. The mean of each bit is now

$$\text{Probability of a 1} + (\text{Probability of a 0} \times \text{Probability of 00}),$$

which translates to

$$\frac{1}{2} + \frac{1}{2} \times \frac{1}{4} = \frac{5}{8}.$$

For two or three bits, this method is fine, but as the lattice gas problem size grow, we need to look at more bits (see Table 2.10). For the largest problem size used, 7 consecutive
bits would need to be looked at to generate each biased random bit. The time to do this biasing is relatively fast for small problem sizes, but would seem to become unacceptable as the number of lattice sites increases and more bits must be examined in sequence. Computationally however, the time needed to bias the bits is independent of the length of the sequence examined. A mask can be used to look at the proper sequence. The mask is a 32-bit value, with the bit values set to 1 if the bits correspond the biasing sequence, and 0 otherwise. For example, if \( N = 2048 \), Table 2.10 shows that 4 consecutive bits in the random number be looked at to determine whether the rightmost bit (of the four) should be flipped in order to bias it. These 4 bits can be examined by bitwise and'ing the random number with the mask 00000000000000000000000000001111. If the result is the value 0, then the 4 rightmost bits of the random number are each 0, and the rightmost bit is flipped to 1. If the result is a value not 0, then the rightmost bit is not flipped. The random number is right shifted 1 bit and the rightmost bit is checked using the same mask.

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \Delta x )</th>
<th>bias</th>
<th>( &lt;a&gt; )</th>
<th>#bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>1/256</td>
<td>1/2</td>
<td>1/4</td>
<td>1</td>
</tr>
<tr>
<td>512</td>
<td>1/512</td>
<td>1/4</td>
<td>1/8</td>
<td>2</td>
</tr>
<tr>
<td>1024</td>
<td>1/1024</td>
<td>1/8</td>
<td>1/16</td>
<td>3</td>
</tr>
<tr>
<td>2048</td>
<td>1/2048</td>
<td>1/16</td>
<td>1/32</td>
<td>4</td>
</tr>
<tr>
<td>4096</td>
<td>1/4096</td>
<td>1/32</td>
<td>1/64</td>
<td>5</td>
</tr>
<tr>
<td>8192</td>
<td>1/8192</td>
<td>1/64</td>
<td>1/128</td>
<td>6</td>
</tr>
<tr>
<td>16384</td>
<td>1/16384</td>
<td>1/128</td>
<td>1/256</td>
<td>7</td>
</tr>
</tbody>
</table>

where: \( c = 1 \)
\( \nu = 2^{-8} \)

Table 2.10: Number of sequential bits needed for biasing

This technique works well only if the bias can be represented as \( \frac{1}{2} + \frac{1}{2^k} \), for some \( k \) (\( k = 3 \) in the above example). If the bias is a less easily represented value, especially an irrational number, a different method must be utilized. The solution was to use a random number generator that had all bits usable as random. Each bit in the number could then be treated as a separately generated random bit. From above, the lagged Fibonacci generator \( LF[55,24,32,+] \) was determined to be an efficiently implemented generator which has good distribution properties, a period long enough to generate the needed amount of random
numbers, and each bit can be considered statistically random. As each random number was generated, it was compared with the bias. If the value was greater than the bias, all the 0 bits in the number were flipped to 1. This new sequence of bits was distributed to the grids, one bit per lattice site. Figure 2.5 shows the times needed to bias all the bits for each problem size $N$ using the three discussed methods and the decrease in time gained by the latter method.

![Figure 2.5: Time needed to bias $10^6$ bits](image)

The manner of bit distribution is important. The straightforward method is a horizontal distribution, or linearly along the grids. The bits are distributed such that the 32 bits from the first random number go to the first 32 lattice sites, the bits from the second random number go to the second 32 sites, the 32 bits from the third random number go the third 32 lattice sites, and so forth. Bits are distributed to all sites on grid 1, then all sites on grid 2, and continuing until all sites on the final grid have received their random bit. This horizontal distribution however causes correlations in the random bits. As Figure 2.6 shows, the correlations manifest themselves as periodic spikes in the solution of the lattice.

---

1 Actually, to eliminate a potential source of correlations, the bit flipping proceeded based on the immediately preceding random number.
CHAPTER 2. RANDOM NUMBER GENERATION

gas. Remembering that the bits are really biased 32 bits—one random number—at a time, it's easy to see how the correlations arise. Biasing is done by selectively flipping 0 bits into 1 bits. When a random number has all of its 0 bits flipped, there are now thirtytwo 1 bits. If these bits are sent to adjacent lattice sites, the effect to have a local advection to the right (i.e. a 'shock') for those thirtytwo sites. It then may take several time-steps for that region to have enough left-moving particles to smooth out the 'shock'. For small lattice gas problems, the bias is large enough that more than $\frac{1}{8}$ of the random numbers will have their 0's flipped and there are never enough 0 bits to allow the left-moving particles to smooth out the many 'shocks'.

![Horizontal distribution of random bits](image)

Figure 2.6: Horizontal distribution of random bits

The correlations arising from distributing the random bits horizontally can be removed by distributing the bits across the grids, not along them. This vertical distribution works by generating a random number, then assigning bit 1 to site 1 on grid 1, bit 2 to site 1 on grid 2, bit 3 to site 1 on grid 3, and so forth, through bit 32 on site 1 on grid 32. A second random number is generated and its bits are assigned to site 2 on grids 1 – 32 in the same order as for the first random number. Bits from the third random number go to site 3. This process continues until all sites on the first 32 grids have received their random bits. The
vertical distribution repeats for the next 32 grids and continues until all lattice sites have received a random bit. Figure 2.7 shows that the solution is free of correlations.

![Figure 2.7: Vertical distribution of random bits](image)

Enough random numbers were generated at each time step to enable every site to have a biased random bit. For each time step, the randomness of the generated numbers and bits was verified using the average, $\chi^2$, distributional, and $m$-tuple tests described above.

### 2.5 Parallel Considerations

Generating random numbers in parallel adds complications. There is a choice of how to generate and distribute the random numbers. Should one processor be used exclusively, generating a random number upon request from the other processors? This technique leads to a severe communication bottleneck at the generating processor if random number generation is a significant part of the total computation [And90]. A communication bottleneck can slow down the computation enough that the gain in speed from the parallelization is lost. If each processor gets a copy of the same random number generator, there is a problem that at some point in the computation, two or more processors will generate large portions
of the same sequence, leading to cross-correlation or long-range correlation [Bro89]. Such correlations have the effect of degrading the statistical randomness of the overall random number generation. If each processor gets a different generator, a difficulty arises in finding enough acceptable random number generators so that each processor has an acceptable sequence to work from. There aren’t enough generators that meet the previously discussed statistical criteria if the number of processors rises above \( \approx 8 \) [APG92].

It might be possible to have a central processor generate a sequence of random bits that are stored either in global memory or offline. Two problems arise with this technique [Bro89]. The first difficulty is creating too much traffic on the processor network, which can drastically reduce the performance gains made by parallel processing. The second problem is that using shared memory for random number storage and distribution both lowers the amount of shared memory available for the remainder of the computation and the processing incurs the usual performance penalties associated with shared memory locations (e.g. spinlocks). That is, the random numbers stored in memory become a critical section in the lattice gas code. Whenever two or processors need a random number at the same time, there is contention for the stored values. Only one processor can access the random numbers in memory, the others must wait. Overall efficiency is degraded since either the system or the application must ensure that only one processor at a time accesses the random numbers, which is a time-consuming amount of overhead. Additionally, the waiting processors do no useful work while they are waiting for their access to the random numbers.

Another aspect of parallel random number generation that must be considered is the reproducibility of the sequence of generated numbers [FHL87]. Here, reproducibility is a parallel concept, it is not the same as asking if a generator will generate the same sequence of values given the same initial value, it instead asks if the sequence of values is the same independent of how the generator is distributed onto the underlying processors. Reproducibility is necessary for debugging purposes, since often in parallel processing the order in which individual processors finish their task is non-deterministic. This non-determinism can cause problems while debugging, as the debugging process itself consumes system resources which may alter the computation’s finishing order. Having a sequence of random numbers which is guaranteed to repeat exactly (given the same seed value) eliminates having to worry whether debugging alters the random number generation.
The method chosen to generate random numbers in parallel is to have a central processor generate a sequence of random numbers that are distributed to the other processors at initialization. The processors then use their given number as a seed into a common, but not shared, random number generator. If the generator is well chosen, each processor generates a subsequence of the full sequence of the generator. The subsequences will show no cross-correlations with each other since the full sequence has no correlations. In addition, this method is independent of the number of physical processors. If the initial seed to the central processor is re-used, the sequence of generated subsequences will be the same.

The lagged Fibonacci generator $\text{LF}[55, 24, 32, +]$ is the common generator. It was seen earlier that the period of this generator is $O(10^{25})$, which is much longer than the $O(10^{12})$ random numbers needed by the lattice gas application. A multiplicative congruential generator was used to generate the initial sequence of the fifty-five values carried by the lagged Fibonacci generator. Each processor thus had its own $\text{LF}[55, 24, 32, +]$ generator. As each processor was fired up, it requested the central processor to send the next fifty-five values as its initial sequence. The seed into the multiplicative congruential generator was saved so that the sequence of generated random numbers could be recomputed. Each processor is guaranteed to reproduce the same random numbers, if desired, since it can be seeded with the same initial fifty-five values.
Chapter 3

Lattice Gas Automata

3.1 Introduction

Lattice gas automata are a somewhat recent method used to solve diffusive partial differential equations. They are cellular automata, that is, computations discrete in the independent variable, as well as in space and time (see Section 2.2.4). Lattice gases have been used to model differential equations such as the diffusion equation [LB90], heat equation [Rod86], and the wave equation [CCDL88]. Lattice gas methods have also been used to examine more complex systems such as flow in porous media [Rot88, Pap93], Navier-Stokes fluids [dLF86, FdH+87, FHP86], advection-diffusion equations [EL93, BB92, BB91], magnetohydrodynamics [CM87], electrostatics [MD87], and immiscible two-phase fluids [RK88]. They represent an alternative to standard finite difference method schemes and have been applied to problems with complex geometries such as the equilibrium and growth morphology of a crystal in a diffusion field [SU90], aerosol droplet coagulation and deformation [DSF91], and transport properties of tracers in a Coulomb gas [PY91]. The stability and Boolean structure of lattice gases is well suited to other particle transport problems such as electron transport and degeneracy effects in semiconductor devices [Anc90].

The general idea is to set up a lattice, on which microscopic particles exist. The particles are only computational, they do not correspond to physical particles. Particles have an associated mass and velocity. Velocity is determined by how far away the neighboring sites are. If the lattice is regular in a geometric sense such that all sites have nearest neighbors equidistant apart (e.g., a square or hexagonal grid in 2 dimensions), mass and velocity are
usually normalized to unity. Particles may also have other associated properties such as temperature or color. The lattice gas evolves by moving the particles, then applying collision rules at each site for many time steps. The particles are allowed to move only along the lattice directions to their nearest neighboring sites. There may be rest particles which don’t move unless a collision with a moving particle occurs. The rules govern the interaction of the incoming particle collisions and determine their outgoing velocities. Some of the rules may be stochastic, that is, particle motion need not be completely deterministic.

Chen [Che93] has noted that "the fluid flows we observe in nature are averages over the trajectories of a very large number of particles, and the details of the particle dynamics have little to do with the observable properties". Lattice gases are a method for implementing a system of particle dynamics that yield macroscopically relevant (i.e. observable) behavior. The power of the lattice gas derives from two processes. The first is conserving various particle properties during collisions. The most common conserved particle properties are: particle number (i.e. mass) and particle velocity and number (i.e. momentum). Other conserved properties include energy, temperature, and color.

The second process is averaging the lattice gas properties to derive corresponding macroscopic, and physically meaningful, properties. It is these averaged, macroscopic, quantities that asymptotically behave as the solution of the partial differential equation would lead one to expect. Averaging can take place with temporal, spatial, or aggregate techniques.

Temporal averaging runs the gas for many micro-time steps, which are then averaged to get a macro-time step. The macro-time steps evolve according to the differential equation. Temporal averaging is best if the system is sensitive to fluctuations in the initial conditions. Spatial averaging sets up a micro-lattice upon which a macro-lattice is overlaid. The macro-lattice evolves by averaging the underlying micro-lattice state at each time step. Aggregate averaging requires the use of multiple grids, each evolving in parallel. Averaging is performed site-by-site across all of the lattices. Aggregate averaging was used throughout all the Burgers’ equation simulations.
3.2 An Automaton for Burgers’ Equation

We will be looking at a lattice gas which models the nonlinear advection-diffusion equation in 1-dimension. Known as Burgers’ equation, the equation is written

\[
\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}.
\]

(Burgers' 1)

Burgers’ equation describes the evolution of shock waves in 1 dimension and is often used to model shallow channel flow. The dissipation constant (ν) governs the speed of the shock formation.

For the Burgers’ Equation lattice gas, a 1-dimensional grid is used (see figure 3.1). The sites are numbered from left-to-right as 0 – N. The grid is periodic such that site_0’s left neighbor is site_N, and conversely, site_N’s right neighbor is site_0.

![Grid for Burgers’ equation lattice gas](image)

Figure 3.1: Grid for Burgers’ equation lattice gas

Particles may only move to the left or to the right; there are no rest particles. All particles move at unit speed and have unit mass and move in lockstep. An exclusion rule exists, which means there can be at most one particle per site in each direction at each timestep. Figure 3.2 shows the four possible valid states at each lattice site. Each of the states is mutually exclusive of the other three states.

A pictorial representation of the collision rules can be seen in Figure 3.3.

The number of particles at each site (i.e. mass) is conserved. That is, particles are not created during a collision, neither are they destroyed. For sites with zero or two particles, the collision rules effectively don’t change the state. This is not the case for sites with only one particle. The velocities upon collision are stochastic, that is, a single particle will rebound (more correctly change direction) probabilistically depending on the dissipation constant ν as shown in the Table 3.1. The bias of the lattice gas is defined to be a. It is this value that the mean of the random bits should equal. This is the model introduced by Boghosian and Levermore [BL87].
The automaton was run for time $0 \leq t \leq 0.25$ seconds, subject to the initial condition

$$u(x, 0) = \frac{(\cos(2\pi x) + 1)}{2}, \ 0 \leq x \leq 2\pi$$

and which has been proven [Elt90] to model Burgers' equation in one dimension. The lattice gas doesn't model Burgers' equation directly. As Elton and Rodrigue have shown [ER90], the equation actually modeled by the lattice gas is

$$\frac{\partial}{\partial t} + c\frac{\partial}{\partial x} A(u) = \nu \frac{\partial^2}{\partial x^2} u$$

Before Collision After Collision Probability

\[
\begin{array}{c|c|c}
\rightarrow \cdot \leftarrow & \leftarrow \cdot \rightarrow & 1 \\
\rightarrow \cdot \leftarrow & \{ \leftarrow \cdot \rightarrow \} & \frac{a}{a} \\
\rightarrow \cdot \leftarrow & \{ \rightarrow \cdot \rightarrow \} & \frac{a}{a} \\
\rightarrow \cdot \rightarrow & \rightarrow \cdot \rightarrow & 1 \\
\end{array}
\]

Figure 3.3: Collision Rules
CHAPTER 3. LATTICE GAS AUTOMATA

### Table 3.1: Collision rules for Burgers' equation lattice gas

<table>
<thead>
<tr>
<th>Before Collision</th>
<th>After Collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>right</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

where: $a = \frac{1 - \nu}{2}$

$\bar{a} = \frac{1 + \nu}{2}$

It isn't at all obvious that the above set of lattice gas rules and definitions model Burgers' equation. A detailed proof is given in appendix A. The lattice gas that has been described seems to have mysteriously sprung from the nether reaches of statistical mechanics. But Boghosian and Levermore [BL87] note that there is a solid and reasonably well-known motivation for the rules of the lattice gas. The authors show that the collision rules for the nonlinear advection-diffusion equation derive from random walk concepts:

A system of particles executing an uncorrelated, unbiased random walk obeys the diffusion equation,

$$\frac{\partial n}{\partial t} = \nu \frac{\partial^2 n}{\partial x^2}$$

where $\nu = (\Delta x)^2/2\Delta t$ is the diffusion coefficient ... and an uncorrelated random walk that is biased so that the probability of a step to the right is $(1 + \bar{a}/2)$, and the probability of a step to the left is $(1 - \bar{a}/2)$ ... leads to linear advection in the direction of the bias. That is, the density of the system obeys

$$\frac{\partial n}{\partial t} + c \frac{\partial n}{\partial x} = \nu \frac{\partial^2 n}{\partial x^2}$$

where $c = \bar{a}\Delta x/\Delta t$ is the linear advection coefficient, and $\nu = (\Delta x)^2/2\Delta t$ is the diffusion coefficient.
CHAPTER 3. LATTICE GAS AUTOMATA

Using a lattice gas of 512 sites and 128 the initial condition is shown in figure 3.4. The evolution of Burgers' equation subject to the above initial condition is shown in figure 3.5.

![Figure 3.4: Initial Condition](image)

Table 3.2 lists the number of random bits necessary to evolve the lattice gas for 0.25 seconds for different numbers of lattice sites (128 grids is assumed for all problem sizes).

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\Delta x$</th>
<th>$\Delta t$</th>
<th>#timesteps</th>
<th>#random bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>1/256</td>
<td>1/512</td>
<td>128</td>
<td>4,194,304</td>
</tr>
<tr>
<td>512</td>
<td>1/512</td>
<td>1/2048</td>
<td>512</td>
<td>33,554,432</td>
</tr>
<tr>
<td>1024</td>
<td>1/1024</td>
<td>1/8192</td>
<td>2048</td>
<td>268,435,456</td>
</tr>
<tr>
<td>2048</td>
<td>1/2048</td>
<td>1/32768</td>
<td>8192</td>
<td>2,147,483,648</td>
</tr>
<tr>
<td>4096</td>
<td>1/4096</td>
<td>1/131072</td>
<td>32768</td>
<td>17,179,869,184</td>
</tr>
<tr>
<td>8192</td>
<td>1/8192</td>
<td>1/524288</td>
<td>131072</td>
<td>137,438,953,472</td>
</tr>
<tr>
<td>16384</td>
<td>1/16384</td>
<td>1/2097152</td>
<td>524288</td>
<td>1,099,511,627,776</td>
</tr>
</tbody>
</table>

where: $\nu = 2^{-8}$

Table 3.2: Table of parameters for Burgers' equation lattice gas

The number of random bits needed for $N$ lattice sites is computed as 1 bit per lattice.
Figure 3.5: Solution at 0.25 sec

site per grid per timestep, or

\[ \#bits = \#timestep \cdot 128 \cdot N. \]

To initialize, or set up the initial conditions of, the lattice gas, an additional

\[ 2 \cdot 128 \cdot N \]

random numbers are required. The algorithm is cubic. That is, as \( N \) is doubled, the amount of work done is 8 times as much. This arises from a factor of 2 increase in the number of lattice sites and another factor of 4 increase in the number of timesteps. The cubic nature of the lattice gas automata necessitates an efficient algorithm. The following section describes in detail the actual implementation of the lattice gas, using bit vectors, to solve Burgers' equation.

### 3.3 A Bit Vector Implementation

The lattice gas is implemented with bit vectors, one vector for the left moving particles, and one for the right. Position \( i \) in each bit vector corresponds to the \( i \)th node in the grid.
A simplified example will help to show the evolution, for the first time step, of the lattice gas.

- Step 1: Initialize the bit vectors

\[
V_R = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}; \quad V_L = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}
\]

- Step 2: Move the Particles. Circular rightshift of \(V_R\) and circular leftshift of \(V_L\). This is the advection step.

\[
\tilde{V}_R = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}; \quad \tilde{V}_L = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}
\]

- Step 3: Determine which nodes have collisions (i.e. which nodes have both a right and left moving particle).
- Step 4: Determine which nodes have only 1 particle.

\[
C = \tilde{V}_R \land \tilde{V}_L = \begin{bmatrix} 1 \land 1 \end{bmatrix} = 1
\]

\[
P = (\tilde{V}_R) \text{XOR} (\tilde{V}_L) = \begin{bmatrix} 1 \text{XOR} 1 \end{bmatrix} = 0
\]

- Step 5: Generate a vector of uniformly distributed random numbers on \([0, 1)\).

\[
R = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \end{bmatrix} = \begin{bmatrix} .437642 \\ .792787 \\ .164733 \\ .639814 \\ .156772 \\ .861479 \\ .379645 \\ .567560 \\ .345612 \end{bmatrix}
\]
- Step 6: Create a vector of random bits. If \( r_i > \text{bias} \) then 1, else 0.

\[
B = R > \text{bias} = \begin{bmatrix}
.437642 \\
.792787 \\
.164733 \\
.639814 \\
.156772 \\
.861479 \\
.379645 \\
.567560 \\
.345612
\end{bmatrix}
\begin{bmatrix}
.625 \\
.625 \\
.625 \\
.625 \\
.625 \\
.625 \\
.625 \\
.625 \\
.625
\end{bmatrix}
\begin{bmatrix}
0 \\
1 \\
0 \\
1 \\
1 \\
1 \\
0 \\
0 \\
1
\end{bmatrix}
\]

- Step 7: Stochastically determine outgoing velocities for single particle nodes using the vector of random numbers.

\[
\dot{V}_R = P \land B = \begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
0 \\
1 \\
0 \\
0 \\
1
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
1 \\
0 \\
0 \\
1 \\
1
\end{bmatrix} = \begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1
\end{bmatrix}
\]

\[
\dot{V}_L = P \land (1 - B) = \begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
0 \\
1 \\
1 \\
0 \\
1
\end{bmatrix}
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
-0 \\
1 \\
1 \\
0 \\
1
\end{bmatrix} = \begin{bmatrix}
0 \\
0 \\
0 \\
1 \\
1 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
\]

\[
\dot{V}_R = P \land B = \begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
0 \\
1 \\
0 \\
0 \\
1
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
1 \\
0 \\
0 \\
1 \\
1
\end{bmatrix} = \begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1
\end{bmatrix}
\]
• Step 8: Compute the updated right and left vectors for this time step.

\[
\begin{bmatrix}
0 \\
1 \\
0 \\
1 \\
0 \\
0 \\
1
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
= \begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 \\
0 \\
1 \\
0 \\
0 \\
0 \\
1
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
= \begin{bmatrix}
1 \\
0 \\
1 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
\]

For subsequent time steps, Step 1 is changed to:

\[
\bar{V}_R = \bar{V}_R \lor C
\]

\[
\bar{V}_L = \bar{V}_L \lor C
\]

The process then continues for the remaining Steps. Note that all operations, except for Steps 5 and 6 are bitwise operations on bit vectors. At Step 5, a floating point random value is generated for each grid site. At Step 6, a floating point comparison is necessary to generate the random bit vector necessary for the final Step. By combining the floating point operations of Steps 5 and 6 into a single efficient bitwise step, the speed of computation would be increased.
CHAPTER 3. LATTICE GAS AUTOMATA

- Step 5a: Generate a single uniformly distributed random number on $[0, 1)$.

$$R = 0.205479$$

- Step 6a: Create a vector of random bits. If $R > \text{bias}$ then $B = R$, else $B = [1]$. That is, if the random number $R$ is greater than the bias then the random bit vector $B$ is equal to the random number $R$. Otherwise, the 0 bits in $R$ will be flipped, and the resulting random bit vector $B$ will consist of all 1s.

$$B = R > \text{bias} = 0.205479 > 0.625 = [1]$$

- Step 7a: Stochastically determine outgoing velocities for single particle nodes using the vector of random numbers.
CHAPTER 3. LATTICE GAS AUTOMATA

$\hat{V}_R = P \land B = \begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
1 & 1 & 1
\end{bmatrix}$

$\hat{V}_L = P \land (1 - B) = \begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0 \\
1 & 1 & 0
\end{bmatrix}$

- Step 8a: Compute the updated right and left vectors for this time step.

$\hat{V}_R = \hat{V}_R \lor C = \begin{bmatrix}
0 & 0 & 0 \\
1 & 0 & 1 \\
1 & 0 & 1 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 0
\end{bmatrix}$
CHAPTER 3. LATTICE GAS AUTOMATA

\[ \vec{V}_{L} = \vec{V}_{L} \vee C = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \vee \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \]

The advection-collision procedure is performed on each individual grid. At the initial and final times \(\text{time}_{0.00}\) and \(\text{time}_{0.25}\) respectively, the grids were aggregate averaged to get the resulting macroscopic behavior. Figure 3.6 shows how the initial conditions evolve under the aggregate averaging, using the 512 site, 128 grid lattice gas above, into the expected curve.

Setting up the initial conditions requires more detail than was described in Step 1. Remember that the lattice gas really models

\[ \frac{\partial}{\partial t} + c \frac{\partial}{\partial x} A(u) = \nu \frac{\partial^2}{\partial x^2} u \]  \hspace{1cm} (3.6)

along the interval \(0 \leq x \leq 1\), which is then transformed into Burgers' equation

\[ \rho_t + \rho \rho_x = \nu \rho_{xx} \]  \hspace{1cm} (3.7)

by the linear transform

\[ \rho = c(1 - 2u). \]  \hspace{1cm} (3.8)

At each lattice site \(x_i\) a random number \(r_{ir}\) is generated. A right-moving particle is placed at \(x_i\) if \(y \geq r_{ir}\), where \(y\) is the transformed initial condition at \(x + (i - 1)\Delta x\). A second random number \(r_{il}\) is generated. If \(y \geq r_{il}\), then a left-moving particle is placed at \(x_i\). The right- and left-moving particles are thus randomly placed along a grid in proportion to the \(y\)-value of the initial condition. In other words, the higher the value of the initial condition at site \(x_i\), the higher the likelihood that there will be a right- and/or left-moving particle at that site. Each grid is initialized independently.
Figure 3.6: Initial condition evolution as the aggregate of grids increases
Chapter 4

Parallel Platforms and Results

4.1 Sequential Results

The lattice gas code and the underlying random number generator were implemented on a Silicon Graphics (SGI) Indigo workstation using the language C. As with most workstations, the SGI has 32-bit integers available. Thus the code for both the lattice gas and the generator was debugged and tested with portability in mind. Timing and statistical data for the random number generators were created on the SGI. Once the code was working, it was ported to the BBN and CRAY multiprocessors.

The random number generator LF[55,24,32,+] was implemented using a circular linked-list to maintain the sequence of lag values. Each grid in the lattice gas was dynamically allocated as an array of integers, one array for left-moving particles and one array for right-moving particles. Each bit in the integers represented a single lattice site. Advection of the particles was performed with arithmetic left- and right-shift operations on the integers. Additional code was required to correctly advect the leftmost and rightmost bits into the adjacent rightmost and leftmost bits of neighboring integers and to correctly wrap the 0th and nth bits.

As seen earlier in Table 3.2, the lattice gas technique is a cubic method. Doubling the number of lattice sites increases the amount of computation eight-fold. In the introduction, it was noted that more than \( \frac{1}{6} \)th (17.9%) of the execution time was spent just generating random numbers. Using one random number to create 32 random bits decreased the percentage of time spent generating the random numbers to 2.5%, or \( \frac{1}{40} \)th of the overall
execution time. This represents an speedup of 7 times. The table below 4.1 is a breakdown of the percentage of execution time spent in each of the phases of the lattice gas. The values were generated from the UNIX utility prof.

<table>
<thead>
<tr>
<th>Phase</th>
<th>1 RN per site</th>
<th>1 RN per 32 sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>collision</td>
<td>78.0</td>
<td>87.3</td>
</tr>
<tr>
<td>generation</td>
<td>17.9</td>
<td>2.5</td>
</tr>
<tr>
<td>initialization</td>
<td>1.2</td>
<td>4.0</td>
</tr>
<tr>
<td>advection</td>
<td>0.8</td>
<td>1.8</td>
</tr>
<tr>
<td>ensemble</td>
<td>0.2</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Table 4.1: Lattice gas execution profile percentages

The lattice gas defined in Section 3.2 models the behavior of an initial condition evolving according to Burgers' equation 1.1. The lattice gas was evolved from time= 0.00 to time= 0.25 seconds subject to the initial condition Equation 3.2. For the smallest problem size of 256 grid points, the initial condition and the solution at 0.25sec are shown in Figures 4.1 and 4.2.

Figure 4.1: Initial condition, 256 Sites
For the largest problem size of 16384 grid points, the initial condition and the solution at 0.25 sec are shown in Figures 4.3 and 4.4.

The lattice gas solution agrees well with a finite difference approximation to Burgers' equation\(^1\). The particular scheme is:

\[
P_{i}^{n+1} - P_{i}^{n} = \frac{\nu \Delta t}{(\Delta x)^2} \left[ P_{i+1}^{n} - 2P_{i}^{n} + P_{i-1}^{n} \right] - \frac{\Delta t}{4\Delta x} \left[ (P_{i+1}^{n})^2 - (P_{i-1}^{n})^2 \right].
\]  

This method is first-order accurate in time and second-order accurate in space. The stability criterion is

\[
\Delta t \leq \frac{(\Delta x)^2}{2\nu}.
\]

Using the transformation

\[
P_{i}^{n} = c(1 - 2U_{i}^{n}),
\]

the solutions of the finite difference method can be compared with those from the lattice gas. The finite difference method was run with \(N = 32768\) points.

---

\(^1\) Burgers' equation can be solved exactly using analytic methods. Applying the Hopf-Cole transform to Burgers' equation transforms it from a non-linear advection-diffusion equation into the heat equation [Sac87], which can then be solved using Fourier analysis or Green's functions techniques.
CHAPTER 4. PARALLEL PLATFORMS AND RESULTS

Figure 4.3: Initial condition, 16384 Sites

Figure 4.4: Solution at 0.25 sec, 16384 Sites
The lattice gas solution is quite noisy, but this is to be expected since the technique is stochastic in nature. The position of the developing shock (near $z = 0.75$) is not a function of the number of lattice sites, it is a function of the parameters $c$ and $\nu$. Increasing $N$ serves to make the lattice solution more accurately handle the sharp transition at the onset of the shock.

### 4.2 Parallel Considerations

One of the aspects which makes parallel processing a non-trivial task is the inherent non-deterministic nature of parallel execution. There is no guarantee as to which order the processors will execute and often no guarantee which processor has what data to work with. Codes must have sections in which the actual order of execution doesn’t affect the results in order to be parallelizable. For example, if there are four processors and a (large) loop that has four iterations, running the code once may yield the following order in which the iterations finish: 1, 2, 3, 4. While such a sequential order is often assumed when writing code, another run might yield the order 2, 4, 3, 1. As long the final results remain unaffected by the particular order of parallel execution, the code can be parallelized.

There are three potential ways to exploit parallelism in the application:

- Run independent tasks in parallel
- Run different iterations of loops in parallel
- Run different sections of a single iteration in parallel.

The lattice gas code has two routines that are executed once (for all the sites and particles) per timestep. The first applies the collision operator to the particles and the second advects the resulting particles to their neighboring sites. Since the advection stage is dependent upon the results of the collision stage, these two processes are not independent. Thus there are no independent tasks which can be run in parallel.

Within either the collision or advection routines, each iteration is dependent on the results from the previous iteration. That is, the state of the lattice gas at a particular time step requires that all previous time steps be completed in sequential order. Therefore the
code cannot be parallelized by running different iterations (i.e. timesteps) of the loops in parallel. Such a type of task loop construct is known as ordered loop execution.

The third method of parallelization is applicable to lattice gas codes. For a single timestep, the iterations within the collision or advection phases of the code may be done in parallel. As long as the indices of the loop are used to explicitly address independent parts of the bit vectors, the loop can execute in any order. Each loop iteration works only on one part of the data and subsequent iterations don’t interfere with the data previously examined. Information that must be shared between sites or different parts of the bit vectors can be exchanged at the end of the loop iteration. Examples of information that must be shared are the bits at the ends of the representative integers—which must be passed to the left- or right-most positions of the neighboring integers, and the distribution of the random bits.

The exchange of information in this manner must occur sequentially to ensure that the exchange is with correct and current data. Guards are used to allow the exchanges to occur in the proper order and without interference. Unfortunately, the use of guards decreases the amount of parallelism. The number of guards and amount of sequential sections of code may be more important when examining how much decrease in execution time is to be gained by running in parallel than the number of available processors or the best optimization of the parallel sections of code. A commonly used rule to determine the maximum gain, or speedup, is given by Amdahl’s law [FJL+88]

\[ S = \frac{1}{\text{seq} + \frac{1-\text{seq}}{N}} \]

where,

- \( S \) Speedup in execution time,
- \( \text{seq} \) percentage of code that is run sequentially,
- \( N \) number of processors available.

Analysis of Amdahl’s law shows that if only 5% of the code is executed sequentially, running on 16 processors can yield a speedup of at most 9.1 (not 16) and 14.3 for 48 processors. A 10% sequential code drops even farther, 6.4 and 8.4 respectively. The distribution of the random bits and the need to pass left- and right-most bits between the representative
integers must be performed sequentially, and will lessen the speedup when the code is run
in parallel.

Lattice gas codes are in principle easy to parallelize. The simplest way is to allow each
lattice node to be a virtual processor [Hie92]. Each processor needs only a small memory
and communicates only with its nearest neighbors. The availability of massively parallel
machines with 16384 nodes is extremely limited and so the lattice gases must be run on
whichever parallel machines are accessible. For this project, parallelism is obtained on two
different types of machines. The BBN TC2000 is an early implementation of a massively
parallel machine. The CRAY C90 is a parallel supercomputer.

4.3 The BBN TC2000

The BBN TC2000 is a multiprocessor with up to 51 physical nodes available for multipro-
cessing. The BBN is a distributed memory machine with 2GB of memory and 18GB of disk
storage. This means there is not a single, shared area of memory that all processors access.
Instead, each processor has its own local memory. The BBN is set up however, so that the
user sees a global memory. The system, or the programmer, keeps track of where the data
actually lay and is responsible for getting the data to where it is needed. The user isn’t
required to know where any specific data reside. A program may act as if the memory really
is global. The processors communicate by means of a static butterfly network [BBN89]. The
butterfly network is a Banyan network similar to that used when fast Fourier transforming
n pieces of data in \( \log_2 n \) stages [HJ88]. Figure 4.5 (from Krishnamurthy [Kri89]) shows the
communication paths for a general 32 node system. Because each node has only two com-
munication channels, the nodes cannot communicate directly with all of the other nodes.
More than one communication step may be necessary to transfer data from one node to
another, requiring that intermediate channels be used to forward the data.

BBN has implemented the butterfly network in such a way that every processor is
able to communicate with each memory directly in one step. Figure 4.6 (from Pawley et
al [PBTC89]) shows that the processors are connected four at a time to 4 \times 4 crossbar
switches, that in turn are connected to similar crossbar switches which connect the proces-
sor memories four at a time. The switch allows a processor to have direct access to any
memory. The memory of the processors collectively form the shared memory of the TC2000.
The global memory seen by the user occurs by allowing the processors to map the same memory segments into their respective address spaces [Cer89] within the operating system.

Communication occurs under user control—there are different paradigms that may be used for inter-processor communication. The two paradigms are known as Split-Join and Message Passing. The split-join concept allows users to assume the computer is a virtual multi-processor. The processors form a team that execute the code from start to finish. Message Passing assumes that each processor has its own local memory and all sharing of data occurs through messages passed between the processors. The message passing model requires that all processes execute the user program from start to finish. They are described more fully below.

The lattice gas code and the random number generator were ported to the BBN and run on a single node with only a few minor changes (such as differing system function calls to obtain timing values). The code was modified separately for the two communication paradigms. When running the code on more than one processor, an environment variable was set which let the system know how many processors the program was to be executed with. When enough processors were available, the scheduler allowed the program to be run.
4.3.1 The Split-Join Model

The split-join model assumes that all processors start at the beginning of the program execution, execute the same code, and remain active until the end of the program. The model is often seen as a team of processors executing the program. Split-join allows a program to be multi-threaded at the highest level. The split-join model has been implemented on LLNL's BBN TC2000 with the Parallel C Preprocessor (PCP). PCP is an extension of the Single-Program-Multiple-Data model [GWBI91]. In PCP, the program is executed on a virtual multiprocessor. The individual team members can be thought of as physical processors and can be programmed as if that is the case. In PCP, a program begins with all of the team members executing the same code. Since the team members must keep a copy of the entire code, there is a loss of available local memory for each processor. It is up to the user to specify which sections of the code are serial and which are parallel. The user must also manage all processor synchronization.

Serial sections are executed with a master block. A master block is a block of code that only one processor executes. Program initialization, I/O, and memory allocation are usually performed within a master block. A master block may also be used to initialize data shared by all team members, such as accumulators. The other processors, known as
the team slaves, do not idle during a master block, they continue to execute the code. In fact, the team slaves execute the code without waiting for the end of the master block. This is an unacceptable situation, as the team slaves might reach the end of the code and cause the program to terminate—note that program termination occurs when any of the team members terminate. Also, the team slaves might continue without having the proper data because the end of the master block may not have been reached. PCP provides a primitive that allows the user to enforce a synchronization upon the processors that ensures the team slaves won’t inadvertently continue to execute the code. Synchronization is performed with the barrier primitive. A barrier requires that all team members wait at the barrier until the preceding work has been completed. Once all the processors, active or not, have arrived at the barrier, the team members can continue executing.

The sections of code which can be executed in parallel are usually those marked by loops. In PCP, the parallel loop construct is the forall loop. The forall loop divides the passes of the loop amongst the team members by interleaving the loop indices within the team. This is known as fine-grained parallelism. A barrier primitive is normally placed at the end of the forall loop to ensure that each team member has finished executing its portion of the loop before the next section of code is executed.

It may be noted that since the indices of the forall loop are interleaved, the value of a shared variable, such as the accumulator that holds the aggregate averaged value for a particular lattice site, is not deterministic. This is due to the fact that the individual team members may access the shared variable in any order and at any time. Just as correct program execution is ensured by the use of the barrier synchronization primitive, correct shared variable evaluation is ensured by the use of the lock and unlock primitives. These two primitives are used to maintain single access to shared variables or critical sections of the code. Each time a processor accesses a critical section, it sets the lock for that section—no other team member can bypass the lock and gain access to the shared variable—and unlocks the critical section when access is no longer required. Once the critical section is unlocked, the other requesting processors compete to determine which one has the next access. The operating system, not PCP, ensures that each team member has an equal chance of gaining access and that every processor will, at some point, get access to the shared variable.

PCP includes functions that give the status of the team being used. That is, how many
processors make up the team and the index of each particular processor. Memory can be allocated either as shared—global and visible to all team members—or private—local and visible only to that team member. The default in PCP is for memory to be shared among all team members.

### 4.3.1.1 Single Node Results

The lattice gas was run on the BBN TC2000 using the PCP split-join model with a single processor. For all problem sizes, the number of grids was held constant at 128. Figure 4.7 shows the execution times as a function of $N$. For both $N = 8192$ and $N = 16384$ lattice sites, the run times were estimated (by using the eight-fold increase in computational effort and multiplying the time for $N = 4096$ by 8 and 64 respectively). The increase in run times for the lesser values of $N$ is indeed nearly 8 as the number of lattice sites is doubled.

![Figure 4.7: Single processor execution times on BBN TC2000 using PCP](image)

### 4.3.1.2 Multi-Processor Results

The lattice gas was run on the BBN TC2000 using the PCP split-join model using multiple processors. Table 4.2 lists the execution times for the matrix of problem sizes and number
of processors.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>4.41s</td>
</tr>
<tr>
<td>512</td>
<td>33.22s</td>
</tr>
<tr>
<td>1024</td>
<td>4.54m</td>
</tr>
<tr>
<td>2048</td>
<td>36.46m</td>
</tr>
<tr>
<td>4096</td>
<td>4.82h</td>
</tr>
<tr>
<td>8192</td>
<td>(38.56h)</td>
</tr>
<tr>
<td>16384</td>
<td>(12.85d)</td>
</tr>
</tbody>
</table>

Table 4.2: Execution times on BBN TC2000 using PCP

For all problem sizes, the number of grids was held constant at 128. For both \( N = 8192 \) and \( N = 16384 \) lattice sites, some of the run times were estimated (by using the eight-fold increase in computational effort and multiplying the time for \( N = 4096 \) by 8 and 64 respectively). The estimated values are shown in parentheses.

It may be seen that the decrease in execution times as the number of processors is increased for a particular problem size is not linear. This effect is most noticeable for the smaller problem sizes. The overhead introduced by the parallel operations in the code limit the gain in speedup—there isn’t enough work to distribute to the processors for small applications. However, as the problem size is increased the decrease in run times by adding processors becomes greater. This is graphically illustrated in figure 4.8.

Another factor that keeps the speedup away from linearity as the number of processors is increased is the communication costs. Increasing the number of nodes increases the amount of data scattering across the memory. This helps slow the execution down because processors must access data residing on other nodes more often; they spend more time waiting for data as the number of processors increase.

### 4.3.2 The Message Passing Model

Message Passing assumes that each processor has its own local memory and all sharing of data occurs through messages passed between the processors. Message Passing puts all inter-processor communication under the control of the programmer. Data is distributed
amongst the nodes by sending messages back-and-forth. The code is constructed so that one thread is the master and all others are considered slaves. The master is given the identifier (ID) 0, and each slave takes on a unique integer ID between 1 and the number of slaves. As the program begins execution, the master performs all initializations and creates the slaves. Data is distributed to each of the slaves from the master by sending a message to a specific slave (using the slave's ID) with the appropriate data. The slaves then begin executing. If a slave needs data residing at another node, it requests the data by sending a message to the other slave—bypassing the master—directly requesting that the other slave send the requested data. Similarly, a slave may send data directly to another node without using the master.

When a slave has finished its task, the master is sent a message that the slave has terminated. The master can then begin a new task on the available node. When all slaves have completed, the master then cleans up and terminates the program. A program can be broken down into parallel and sequential sections. Parallel sections are run as elaborated above. Sequential sections must not be performed in parallel. The master must wait until all executing slaves have sent termination messages before it proceeds into the sequential

Figure 4.8: Multi-processor execution times on BBN TC2000 using PCP
LLNL has implemented Message Passing on the BBN TC2000 with the Livermore Message Passing System (LMPS). LMPS is a message passing interface modeled after the Argonne Message Passing System [Wel91]. LMPS consists of a C preprocessor and directives which implement the library routines that perform all communication activity. LMPS introduces into the code specific function calls which handle the message passing and create the master and slave tasks. LMPS features include message typing, synchronous and asynchronous message passing, selective reception of messages, blocking receive, and blocking and non-blocking send. No multitasking is allowed on a single node. The LMPS model requires that all processes execute the user program from start to finish.

The general form of a message from one process to another is

\[
\text{request(pid, type, message, length, ack)},
\]

where

- \text{pid} \quad \text{which process is to respond to the request}
- \text{type} \quad \text{what type of message is being passed (i.e. data or control)}
- \text{message} \quad \text{the actual message being passed}
- \text{length} \quad \text{the length of the message}
- \text{ack} \quad \text{should requestor block (wait) until responder acknowledges}

The specific requests are SEND and SENDR—transfer data to another process; RECEIVE and SRECEIVE—block until the arrival of a message satisfying a user defined condition; RECVNW and SRECVNW—non-blocking receipt of a message; and BROADCAST—allow one process to send a message to all other processes. In addition, there are routines which provide for initialization and termination of processes and process identification and environment routines. To execute LMPS code, the code must be linked to the PCP library routines described previously in order to properly provide for runtime startup and barrier routines. This means that PCP routines may be used in combination with LMPS routines, although this was not done for the thesis.
4.3.2.1 Single Node Results

The lattice gas was run on the BBN TC2000 using the LMPS message passing model with a single processor. For all problem sizes, the number of grids was held constant at 128. Figure 4.9 shows the execution times as a function of $N$. For both $N = 8192$ and $N = 16384$ lattice sites, the run times were estimated (by using the eight-fold increase in computational effort and multiplying the time for $N = 4096$ by 8 and 64 respectively). The increase in run times for the lesser values of $N$ is indeed nearly 8 as the number of lattice sites is doubled.

![Figure 4.9: Single processor execution times on BBN TC2000 using LMPS](image)

4.3.2.2 Multi-Processor Results

The lattice gas was run on the BBN TC2000 using the LMPS message passing model using multiple processors. Table 4.3 lists the execution times for the matrix of problem sizes and number of processors.

For all problem sizes, the number of grids was held constant at 128. For both $N = 8192$
and $N = 16384$ lattice sites, some of the run times were estimated (by using the eightfold increase in computational effort and multiplying the time for $N = 4096$ by 8 and 64 respectively). The estimated values are shown in parentheses.

It may be seen that the decrease in execution times as the number of processors is increased for a particular problem size is not linear. This effect is most noticeable for the smaller problem sizes. The overhead introduced by the parallel operations in the code limit the gain in speedup—there isn’t enough work to distribute to the processors for small applications. However, as the problem size is increased the decrease in run times by adding processors becomes greater. This is graphically illustrated in figure 4.10.

Another factor that keeps the speedup away from linearity as the number of processors is increased is the communication costs. Increasing the number of nodes increases the amount of data scattering across the memory. This helps slow the execution down because processors must access data residing on other nodes more often; they spend more time waiting for data as the number of processors increase.

4.4 Multitasking on the CRAY C90

The CRAY C90 is the latest in the CRAY Y-MP series of vector supercomputers. It has 16 processors and 2GB of memory. Unlike the memory of the BBN, the CRAY memory is a single, global memory. Each processor has the same access time to any part of the memory. Although there is a C compiler on LLNL’s CRAY, parallel code needs to be written in

<table>
<thead>
<tr>
<th>N</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>4.01s</td>
<td>2.46s</td>
<td>1.37s</td>
<td>1.07s</td>
<td>1.03s</td>
<td>1.01s</td>
<td>1.00s</td>
<td>1.00s</td>
</tr>
<tr>
<td>512</td>
<td>31.86s</td>
<td>18.96s</td>
<td>10.73s</td>
<td>7.97s</td>
<td>7.27s</td>
<td>6.85s</td>
<td>6.62s</td>
<td>6.48s</td>
</tr>
<tr>
<td>1024</td>
<td>4.28m</td>
<td>2.46m</td>
<td>1.42m</td>
<td>1.01m</td>
<td>0.88m</td>
<td>0.80m</td>
<td>0.76m</td>
<td>0.74m</td>
</tr>
<tr>
<td>2048</td>
<td>34.02m</td>
<td>18.90m</td>
<td>11.15m</td>
<td>7.56m</td>
<td>6.38m</td>
<td>5.67m</td>
<td>5.31m</td>
<td>5.07m</td>
</tr>
<tr>
<td>4096</td>
<td>4.60h</td>
<td>2.49h</td>
<td>1.49h</td>
<td>0.96h</td>
<td>0.79h</td>
<td>0.69h</td>
<td>0.64h</td>
<td>0.61h</td>
</tr>
<tr>
<td>8192</td>
<td>(36.83h)</td>
<td>(19.92h)</td>
<td>11.88h</td>
<td>7.37h</td>
<td>5.85h</td>
<td>5.05h</td>
<td>4.63h</td>
<td>4.41h</td>
</tr>
<tr>
<td>16384</td>
<td>(12.27d)</td>
<td>(6.64d)</td>
<td>(3.96d)</td>
<td>(2.46d)</td>
<td>(1.95d)</td>
<td>(1.68d)</td>
<td>(1.54d)</td>
<td>(1.40d)</td>
</tr>
</tbody>
</table>

Table 4.3: Execution times on BBN TC2000 using LMPS
Figure 4.10: Multi-processor execution times on BBN TC2000 using LMPS

**FORTRAN** to be in order to more easily access all the directives and routines. Rewriting the code into **FORTRAN** was fairly straightforward. The rewritten code was then ported to the CRAY. CRAY has available 64-bit words. This caused a minor problem since the random number generator used 32-bit words. Rather than develop a new generator that used words of length 64, it was decided to use two 32-bit random numbers in tandem to create one 64-bit random number for the CRAY. With this method, the same sequence of random numbers would be generated on all the platforms, and the amount of numbers would be the same also.

Parallelism on the CRAY occurs using *multitasking*. Multitasking allows the user to exploit different levels of parallelism in the code. *Autotasking* puts the control of scoping and parallelism in the compiler and the strategies in its optimization preprocessors [Syn91]. The typical autotasked code has a *master* processor which executes the code before a loop. The other processors are *slaves*, which execute the loops. The use of **PARALLEL DO** or **DO ALL** directives is the user equivalent of what the compiler adds to the code [Nag90].

---

2 More recent versions of the C compiler allow microtasking and autotasking using *pragmas*. The compiler first runs the code through a preprocessor which translates the pragmas into the same system calls used by the **FORTRAN** compiler. Macrotasking is done by calling the **FORTRAN** routines directly.
Autotasking is a good first step in developing parallel codes on the CRAY. Autotasking is useful because little code revision is necessary. Unfortunately however, the compiler doesn’t often find the best parallel organization [Dec93] for the code since it mainly checks for loops which can be parallelized. Two user controlled methods are known as microtasking and macrotasking.

Microtasking is very similar to the Split-Join model described in Section 4.3.1. All the processors execute the same code. They all begin at the top of the code and continue until one processors reaches the bottom. As with PCP, there are directives that specify which sections of the code can be executed in parallel [YW93]. Parallel sections can are indicated with the CMIC$ DO GLOBAL directive or the CMIC$ PROCESS–CMIC$ ALSO PROCESS–CMIC$ END PROCESS primitives. The compiler adds GOTO statements which act as barriers to synchronize the processors at the end of the parallel and sequential sections. GUARD and END GUARD statements can (and should) be used instead. Just as with PCP, since each processor executes the same code, there is a local memory cost due to the code redundancy. The gain with respect to autotasking is the ability to execute blocks of code independently and in parallel which are difficult to organize as DO loops. Microtasking and autotasking primitives can coexist within a program but not within a subroutine.

While autotasking and microtasking are considered fine-grained parallelism, macrotasking is used to obtain coarse-grained parallelism. The code must be explicitly partitioned into individual tasks. This is referred to as user controlled scoping. Macrotasking can be used in conjunction with both microtasking and autotasking. One can envision each subroutine as a task that can be executed in parallel. The main code could have different subroutines executing concurrently on different pieces of data.

The lattice gas application has no independent tasks so macrotasking was not used. Parallelism can be exploited only within each loop. Each bit and subsequent integer operation can be computed independently, excepting the passing of the right- and left-most bits between the integers. This loop-wise parallelism was implemented using microtasking. Since each bit is dependent on the collision and advection phases of the neighboring bits at the previous timestep, there is no inter-loop parallelism. Autotasking was used to fine-tune the microtasking.
4.4.1 Single Node Results

The lattice gas was run on the CRAY C90 using the multitasking model with a single processor. The code was first subjected to the autotasking compiler and then microtasking directives were added. For all problem sizes, the number of grids was held constant at 128. Figure 4.11 shows the execution times as a function of $N$. For both $N = 8192$ and $N = 16384$ lattice sites, the run times were estimated (by using the eight-fold increase in computational effort and multiplying the time for $N = 4096$ by 8 and 64 respectively). The increase in run times for the lesser values of $N$ is indeed nearly 8 as the number of lattice sites is doubled.

![Figure 4.11: Single processor execution times on CRAY C90](image)

4.4.2 Multi-Processor Results

The lattice gas was run on the C90 using the autotasking model using multiple processors. Table 4.4 lists the execution times for the matrix of problem sizes and number of processors.

For all problem sizes, the number of grids was held constant at 128. For both $N = 8192$
Table 4.4: Execution times on CRAY C90

<table>
<thead>
<tr>
<th>N</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>0.54s</td>
</tr>
<tr>
<td>512</td>
<td>4.27s</td>
</tr>
<tr>
<td>1024</td>
<td>33.86s</td>
</tr>
<tr>
<td>2048</td>
<td>4.52m</td>
</tr>
<tr>
<td>4096</td>
<td>35.95m</td>
</tr>
<tr>
<td>8192</td>
<td>4.84h</td>
</tr>
<tr>
<td>16384</td>
<td>(1.61d)</td>
</tr>
</tbody>
</table>

and \( N = 16384 \) lattice sites, some of the run times were estimated (by using the eight-fold increase in computational effort and multiplying the time for \( N = 4096 \) by 8 and 64 respectively). The estimated values are shown in parentheses.

It may be seen that the decrease in execution times as the number of processors is increased for a particular problem size is not linear. This effect is most noticeable for the smaller problem sizes. The overhead introduced by the parallel operations in the code limit the gain in speedup—there isn’t enough work to distribute to the processors for small applications. However, as the problem size is increased the decrease in run times by adding processors becomes greater. This is graphically illustrated in figure 4.12.
Figure 4.12: Multi-processor execution times on CRAY C90
Chapter 5

Conclusions

5.1 Summary

A lattice gas automaton was implemented on a massively parallel machine (the BBN TC2000) and a vector supercomputer (the CRAY C90). The automaton models Burgers' equation

\[ \rho_t + \rho \rho_x = \nu \rho_{xx} \]

in 1 dimension. The lattice gas evolves by advecting and colliding pseudo-particles on a 1-dimensional, periodic grid. The specific rules for colliding particles are stochastic in nature and require the generation of many billions of random numbers to create the random bits necessary for the lattice gas. The goal of the thesis was to speed up the process of generating the random bits and thereby lessen the computational bottleneck of the automaton.

The first phase was to implement a random number generator that generated a sequence of random numbers over a long period. Four types of generators were examined: multiplicative congruential, shift register, lagged Fibonacci, and a \( k = 2, r = 1 \) cellular automaton. The numbers generated by the random numbers were statistically random based on average measurements. But the multiplicative congruential and shift register generators were seen to have a period that was inadequate for use in the lattice gas implementations. The cellular automaton generators had a period of the same order as the number of random numbers needed by the largest lattice gas problem. In comparison with the lagged Fibonacci generator, the cellular automaton generator was roughly three times as slow. The
lagged Fibonacci generator $LF[55,24,32,+]$ was chosen to be implemented as the basis for the random bit generation.

The random numbers generated by $LF[55,24,32,+]$ were subjected to several $\chi^2$ tests. The generator passed the unordered and ordered $\chi^2$ tests, in the sense of generating uniformly distributed numbers throughout the length of the sequence used by the lattice gas. Passing the tests means that the computed $\chi^2$ values had a level of significance that was neither too random nor too random enough. The generator also passed such near-term distributive $\chi^2$ tests as the frequency, poker, runs above/below, and 2-term serial correlation coefficient tests. Using the $m$-tuple test developed by Marsaglia [Mar85] to measure the randomness of the individual bits within the generated random numbers, each of the 32 bits in the generator was determined to be statistically random. The lagged Fibonacci generator was also shown to be nearly four times as efficient as a well-known random bit generator and thus was determined to be the best random number generator for the lattice gas. That is, $LF[55,24,32,+]$ was found to be a random number number generator that produces a sequence of statistically acceptable random numbers with a period greater than $10^9$, and whose bits may be used as individual random bits.

After the $LF[55,24,32,+]$ generator was implemented to generate a stream of random bits, the bits needed to be biased. The bits needed to be biased in order for the initial sinusoidal condition to advect and the shock front to develop. That is, the mean of the bits was not $\frac{1}{2}$, but equal to a computed value slightly greater than $\frac{1}{2}$ based on the number of lattice sites. The bits were biased in groups of 32 (i.e. one random number at a time) based on the value of the previous random number and distributed in a non-intuitive manner. This had the effect of generating bits with the proper mean values, but without the correlations that arose if the bits were improperly distributed.

The ability to generate biased random bits in this unique manner allowed the lattice gas to be more efficiently implemented. The resulting code spent less than $\frac{1}{4}$th the time generating random numbers as previously. The overall speedup of the code was approximately 4 times. The extra overhead required to access each bit of a generated random number and flip all the 0 bits when called for caused the speedup to be less than ideal. An additional factor that lessened the speedup is that some work must be performed in order to distribute the bits so that correlations don’t affect the automaton. This method of generating random
bits is six times faster than generating one random number and determining if a bit should then be set to 0 or 1. It is also faster than biasing bits based on the patterns of sequentials bits within the random number, as is often done.

The lattice gas was implemented on a workstation first to verify that the bit biasing technique worked—i.e. the lattice gas was more efficient using the new method. The results of the simulations were compared to the solution from a finite difference approximation to Burgers’ equation.

The lattice gas automaton was then ported to the BBN. Memory is distributed to all the processors, but the user sees only a single, global memory. Two different paradigms were used to parallelize the code. The Split-Join model effectively used the processors as a team that executed the code together. Each team member executes the same code, but on different parts of the data. Communication is handled by the system; the user handles all synchronization. The Message-Passing model lets each processor execute its own code. Data is communicated among the processor using explicit messages for requesting and sending. The user handles all communication.

The Split-Join model is programmed using the PCP directives and routines. PCP is not too difficult to program. The sequential code doesn’t need to be modified very much in order to be run in parallel. The Message-Passing model is programmed using the LMPS directives and routines. LMPS is implemented on top of the PCP routines. LMPS is more difficult to program than PCP, but the resulting execution times are faster (≈6–10%). In addition, speedup with LMPS is slightly better (≈3–5%).

For both models, the simulation was run for all problem sizes (although some of the largest problem runs were estimated) for different numbers of processors. Adding processors produced speedup in the execution times, but not a linear speedup. This effect is most noticeable for the smaller problem sizes. The overhead introduced by the parallel operations in the code limit the gain in speedup—there isn’t enough work to distribute to the processors for small applications. However, as the problem size is increased the decrease in run times by adding processors becomes greater until adding more processors shows little gain in speedup, as expected by Amdahl’s law. Another factor that keeps the speedup away from linearity as the number of processors is increased is the communication costs. Increasing the number of nodes increases the amount of data scattering across the memory. This helps
slow the execution down because processors must access data residing on other nodes more often; they spend more time waiting for data as the number of processors increase.

The lattice gas was also ported to the Cray C90. There is a global memory; the processors have equal access time to memory. Parallelism is provided through the multitasking paradigm. As there were no independent task in the application, no macrotasking was used. Only microtasking (and autotasking) was used.

The simulation was run for all problem sizes (although some of the largest problem runs were estimated) for different numbers of processors. Adding processors produced speedup in the execution times, but not a linear speedup. This effect is most noticeable for the smaller problem sizes. The overhead introduced by the parallel operations in the code limit the gain in speedup—there isn’t enough work to distribute to the processors for small applications. However, as the problem size is increased the decrease in run times by adding processors becomes greater until adding more processors shows little gain in speedup, as expected by Amdahl’s law.

### 5.2 Future Work

Optimizing the bit access sections of code used to correctly pass the leftmost and rightmost bits to the respective neighboring sites may help increase the efficiency of the technique. Finding a faster way to distribute the bits should also speedup the simulations. In both cases, since bits are being used, writing the relevant sections in assembly language, appropriate to each machine, should be investigated.

Another area that should be examined is the generation of 64-bit random numbers. Most workstation and larger computers are still 32-bit word based, but machines with 64-bit words are becoming more available. Use of the larger word size would mean that the number of random numbers needed to generate random bits would be halved. In addition, the number of bit access needed above would be halved also. Large word-size machines could make the bit generation method more efficient since less overhead would be required during the various bit operations.

It was shown in Section 2.5 that by using a multiplicative congruential generator to seed the initial sequence of values for the Lagged Fibonacci generators and distributing the values fiftyfive at a time to each processor provided reproducibility to the random
number generation. The reproducibility held only for each specific number of processors. The next phase of investigation for the parallel generation of random numbers involves making the generation totally reproducible. It would be advantageous to be able to create a sequence of values to be independent of the number of processors used. This would allow the computation to run first in a sequential environment. Then using the same seed value to the generator, the code could be run on multiple processors. The sequential result could then be compared with the parallel result. Since the random numbers in the computation were generated in the same order, the solutions should be the same. If the generator was reproducible in this additional sense, there would be a guarantee that the parallel and sequential results would be the same. Multiplicative congruential generators have been demonstrated [KL89, DMP88] that has this property. But as seen earlier, such a generator is unacceptable because its period is too short for use with large lattice gas problems. More importantly, it cannot be used as the basis for most efficiently generating random bits. Perhaps a way can be found to distribute random numbers from a Lagged Fibonacci generator that would allow the same sequence of values to be generated regardless of the number of processors available.

More work needs to be done to determine if the parallel versions of the simulation can be made more efficient. That is, can the speedup in execution time be made closer to linear with increasing processors? On the BBN the speedup decreased to $\approx 0.6$ for the higher number of nodes (for the larger problem sizes). The two paradigms on the BBN need to be examined to determine whether one is more efficient at parallelization. A better understanding of the data scattering may be worthwhile also. On the CRAY, it would be worthwhile to look into using vectorization. Since the lattice grids are implemented as an array of integers, for the larger problem sizes, pushing the arrays through the vector pipes may add some speedup to the overall execution. A disadvantage to this may be the bit access operations, which might throw off the pipelining that makes vectorization so effective.

The method used to generated random bits should decrease the execution time of lattice gas automata. Since some of the collision rules are probabilistic, speeding up the roll of the dice should speed up the simulation. If the random bits don't have to be biased, then the method should be even more effective since the extra computation necessary to bias the bits won't be needed. Additional savings would be gained if the random bits can be
distributed directly, i.e. if correlations between the random bits aren’t a problem then the method becomes more efficient since no extra effort is required to distribute the bits.
Appendix A

Chapman-Enskog Expansion

The lattice gas described in section 3.2 which behaves according to Burgers’ equation

$$\rho_t + \rho \rho_z = \nu \rho_{zz}$$

has the collision rules listed in the table below. By applying the Chapman-Enskog analysis [Bog90] to the collision rules, it will be shown that Burgers’ equation is a description of the lattice gas.

<table>
<thead>
<tr>
<th>Before Collision</th>
<th>After Collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>left right</td>
<td>left right probability</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1 a</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1 a</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

where: $a = \frac{1-\nu}{2}$

$\bar{a} = \frac{1+\nu}{2}$

For a particular lattice site $z$ at time $t$, the particle motions to the neighboring right and left lattice sites ($z + 1$ and $z - 1$ respectively) at the next time step ($t + \Delta t$) are described by the logical equations

$$b_+(x + l, t + \Delta t) = b_r \land (b_+ \oplus b_-) + b_+ \land b_-$$

$$b_-(x - l, t + \Delta t) = \bar{b}_r \land (b_+ \oplus b_-) + b_+ \land b_-.$$  \hspace{1cm} (A.1)

Equivalently, the above equations can be written in algebraic form as

$$b_+(x + l, t + \Delta t) = b_r(b_+ + b_- - 2b_+b_-) + b_+b_-$$

$$b_-(x - l, t + \Delta t) = (1 - b_r)(b_+ + b_- - 2b_+b_-) + b_+b_-.$$  \hspace{1cm} (A.2)
where \( b_r \) is a random bit. The probability of \( b_r \) being a 1 or a 0 is
\[
p(1) = \frac{1+a}{2}, \quad p(0) = \frac{1-a}{2}
\]
respectively. The average of \( b_r \) and its complement \( \overline{b_r} \) is respectively
\[
< b_r >= \frac{1+a}{2}, \quad < \overline{b_r} >= \frac{1-a}{2}.
\]

The Chapman-Enskog analysis requires that correlations between colliding particles are negligible, i.e.
\[
< b_+ b_+ >= < b_+ > < b_+ >.
\]

The random bit is also assumed to be uncorrelated with the dependent variable
\[
< b_r b_+ > = < b_r > < b_+ >
\]
\[
< b_r b_- > = < b_r > < b_- >.
\]

The collision rules A.2 are ensemble averaged, using any of the three averaging techniques (temporal, spatial, and aggregate) described in section 3.1. The ensembled average of a right- or left-moving particle is defined respectively as
\[
n_+ = < b_+ >, \quad n_- = < b_- >.
\]

Ensemble averaging the right-moving particles at site \( x \) at time \( t \) yields
\[
< b_+(x+l,t+\Delta t) > = < b_r(b_+ + b_- - 2b_+b_-) + b_+b_- >
\]
\[
= < b_r(b_+ + b_- - 2b_+b_-) > + < b_+b_- >
\]
\[
= < b_r b_+ + b_r b_- - 2b_r b_+ b_- > + < b_+b_- >
\]
\[
= < b_r b_+ > + < b_r b_- > - < 2b_r b_+ b_- > + < b_+b_- >
\]
\[
= < b_r > < b_+ > + < b_r > < b_- > - 2 < b_r > < b_+ > < b_- > + < b_+ > < b_- >. \tag{A3}
\]

Substituting in the definitions for \( n \) and \( b_r \) above, produces
\[
n_+(x+l,t+\Delta t) = \left( \frac{1+a}{2} \right) n_+ + \left( \frac{1-a}{2} \right) n_- - 2 \left( \frac{1+a}{2} \right) + n_+n_-
\]
\[
= \left( \frac{1+a}{2} \right) n_+ + \left( \frac{1-a}{2} \right) n_- - an_+n_- \tag{A4}
\]

Rearranging the first term yields
\[
n_+(x+l,t+\Delta t) = n_+ - \left( \frac{1-a}{2} \right) n_+ + \left( \frac{1+a}{2} \right) n_- - an_+n_- \tag{A5}
\]

Defining the latter three terms as the collision operator \( C_+ \)
\[
C_+ = - \left( \frac{1-a}{2} \right) n_+ + \left( \frac{1+a}{2} \right) n_- - an_+n_-,
\]
A discrete analog of the Boltzmann equation is derived

\[ n_+(x + l, t + \Delta t) = n_+ + C_+ . \]  

(A.6)

In a similar manner, the discrete analog of the Boltzmann equation is derived for the left-moving particles

\[ < b_-(x - l, t + \Delta t) > = < (1 - b_r)(b_+ + b_- - 2b_+ b_-) + b_+ b_- > \]
\[ = < (1 - b_r)(b_+ + b_- - 2b_+ b_-) > + < b_+ b_- > \]
\[ = < b_+ + b_- - 2b_+ b_- - b_r b_+ - b_r b_- + 2b_r b_+ b_- > + < b_+ b_- > \]
\[ = < b_+ + b_- > + < b_+ > - < b_r b_- > + < b_r b_+ > - < b_r b_+ > + < b_r b_- > \]
\[ = < b_+ > + < b_- > - 2 < b_+ > < b_- > - < b_r > < b_+ > - < b_+ > < b_- > + < b_+ > < b_+ > + b_- > (A.7) \]

\[ n_-(x - l, t + \Delta t) = n_- + n_- - 2n_+ n_- - \left( \frac{1 + a}{2} \right) n_+ - \]
\[ \left( \frac{1 - a}{2} \right) n_- + 2 \left( \frac{1 + a}{2} \right) n_+ n_- \]
\[ = \left( \frac{1 - a}{2} \right) n_+ + \left( \frac{1 - a}{2} \right) n_- + an_+ n_- \]  

(A.8)

\[ n_-(x - l, t + \Delta t) = n_- - \left( \frac{1 + a}{2} \right) n_- + \left( \frac{1 - a}{2} \right) n_+ + an_+ n_- \]  

(A.9)

\[ n_-(x - l, t + \Delta t) = n_- + C_- \]  

(A.10)

The discrete Boltzmann equations for the right- (equation A.6) and left-moving (equation A.10) are called the advection operator.

The lattice gas has one conserved quantity: particle number, which is called \( \Gamma \). Collisions can neither destroy nor create particles. The number of particles at site \( x \) is unchanged by collisions, only their directions (velocities) may change. The collision operators for the right- and left-moving particles must therefore equal zero

\[ 0 = C_- + C_+. \]  

(A.11)

The conserved quantity \( \Gamma \) can now be defined in terms of the advection operator (equations A.6 and A.10)

\[ \Gamma = n_- + C_- + n_+ + C_+ \]
\[ = n_- + n_+. \]  

(A.12)
Equation A.12 is referred to as the conservation of mass equation for the lattice gas. Instead of writing out all the $n_\pm$ and $C_\pm$ terms, the vector notation

\[ n \equiv \begin{pmatrix} n_- \\ n_+ \end{pmatrix} \quad (A.13) \]

\[ C \equiv \begin{pmatrix} C_- \\ C_+ \end{pmatrix} \quad (A.14) \]

is introduced. A direction vector

\[ c_i = \begin{pmatrix} -1 \\ +1 \end{pmatrix} \quad (A.15) \]

is also defined, where $c_1$ is the left-moving direction of $x$ and $c_2$ the right-moving direction. The advection operator (equations A.6 and A.10) can then be written in vector form as

\[ n(x + IC_i, t + \Delta t) = n(x, t) + C(n). \quad (A.16) \]

The vector form of the advection operator is then Taylor series expanded in both $x$ and $t$

\[ n(x + IC_i, t + \Delta t) = n(x, t) + \Delta t \frac{\partial}{\partial t} n + Ic_i \cdot \frac{\partial}{\partial x} n + \mathcal{O}(\Delta x^2, \Delta t^2). \quad (A.17) \]

The Taylor expansion A.17 is substituted into the right hand side of the advection operator A.16 and the second, and higher, order terms are dropped to yield the conservation equation

\[ \Delta t \frac{\partial}{\partial t} n + Ic_i \cdot \frac{\partial}{\partial x} n = C(n). \quad (A.18) \]

To solve the Taylor expansion of the discrete Boltzmann equation, both $n$ and $C$ must first be expanded in an asymptotic series about the (small) expansion parameter $\epsilon$. For the ensemble averaged particles the resulting expansion is

\[ n = n^{(0)} + \epsilon n^{(1)} + \epsilon^2 n^{(2)} + \ldots, \quad (A.19) \]

while for the collision operator, the expansion results in the equation

\[ C = C^{(0)} + \epsilon C^{(1)} + \epsilon^2 C^{(2)} + \ldots. \quad (A.20) \]

The second and higher order terms are dropped to get

\[ n = n^{(0)} + \epsilon n^{(1)} \quad (A.21) \]

and

\[ C = C^{(0)} + \epsilon C^{(1)}. \quad (A.22) \]
Looking at the collision operator for the right-moving particles, and assuming \( a \) is the same order as \( \epsilon \) (i.e. \( a \approx O(\epsilon) \)), the expansion \( A.22 \) results in

\[
C_+ = C_+^{(0)} + \epsilon C_+^{(1)} = C_+^{(0)} + (1 - a)n_+ + \left( \frac{1 + a}{2} \right)n_- + an_+n_-
\]

\[
= \frac{1}{2}n_+ - \frac{1}{2}n_+ + a_2n_- + \frac{a}{2}n_+ + \frac{a}{2}n_+ + an_+n_-
\]

\[
\Rightarrow \quad C_+^{(0)} = \frac{1}{2}(n_- - n_+) \quad (A.26)
\]

\[
C_+^{(1)} = \frac{1}{2}n_+ + \frac{1}{2}n_+ - n_+n_-
\]

Looking at the collision operator for the left-moving particles, and assuming \( a \) is the same order as \( \epsilon \) (i.e. \( a \approx O(\epsilon) \)), the expansion \( A.22 \) results in

\[
C_- = C_-^{(0)} + \epsilon C_-^{(1)} = C_-^{(0)} + (1 + a)n_- + \left( \frac{1 - a}{2} \right)n_+ + an_+n_-
\]

\[
= \frac{1}{2}n_- + \frac{1}{2}n_+ - \frac{a}{2}n_+ + \frac{a}{2}n_+ + an_+n_-
\]

\[
\Rightarrow \quad C_-^{(0)} = \frac{1}{2}(-n_- + n_+) \quad (A.31)
\]

\[
C_-^{(1)} = -\frac{1}{2}n_- - \frac{1}{2}n_+ + n_+n_-
\]

At equilibrium, there are no collisions. The zero'th order of the collision operator can then be equated to zero

\[
0 = C^{(0)} = C_+^{(0)} = C_-^{(0)} = \frac{1}{2}(n_+^{(0)} - n_-^{(0)}).
\]

This implies

\[
n_+^{(0)} = n_-^{(0)} \quad (A.33)
\]

From the conservation of mass equation (equation A.12) and the zero-order collision operator (equation A.33), the zero-order solution for \( n_\pm(0) \) is found to be the conserved quantity \( \Gamma \)

\[
n_+^{(0)} = n_-^{(0)} = \Gamma \quad (A.34)
\]

The Jacobian of the collision operator is defined as

\[
J = \begin{pmatrix}
\frac{\partial C_+^{(0)}}{\partial n_+} & \frac{\partial C_-^{(0)}}{\partial n_-} \\
\frac{\partial C_-^{(0)}}{\partial n_+} & \frac{\partial C_+^{(0)}}{\partial n_-}
\end{pmatrix}
\]

\[
(A.35)
\]
APPENDIX A. CHAPMAN-ENSKOG EXPANSION

\[
= \left( \begin{array}{cc}
-\frac{1}{2} & +\frac{1}{2} \\
+\frac{1}{2} & -\frac{1}{2}
\end{array} \right). \tag{A.36}
\]

The Jacobian has eigenvalues

\[
\lambda^{(0)} = 0 \tag{A.37}
\]

\[
\lambda^{(1)} = -1 \tag{A.38}
\]

and corresponding eigenvectors

\[
\gamma = \left( \begin{array}{c}
+1 \\
+1
\end{array} \right) \tag{A.39}
\]

\[
\xi = \left( \begin{array}{c}
+1 \\
-1
\end{array} \right). \tag{A.40}
\]

The first-order solution for \( n^{(1)}_\pm \) is found by solving the system

\[
J \cdot n^{(1)} = \left[ l \frac{\partial \Gamma}{\partial x} - a \Gamma (1 - \Gamma) \right] \xi \tag{A.41}
\]

which has the solutions

\[
n^{(1)}_\pm = \mp l \frac{\partial \Gamma}{\partial x} \pm a \Gamma (1 - \Gamma). \tag{A.42}
\]

Substituting the zero’th- and first-order solutions (equations A.34 and A.42) into the Taylor series expansion of the advection operator (equation A.18) the hydrodynamic equation for the lattice gas is then

\[
\frac{\partial \Gamma}{\partial t} + a l \Delta t (1 - 2 \Gamma) \frac{\partial \Gamma}{\partial x} = \frac{l^2 \partial^2 \Gamma}{2 \Delta t \partial x^2}. \tag{A.43}
\]

Using the linear transformation

\[
u \equiv \frac{a l}{\Delta t} (1 - 2 \Gamma), \tag{A.44}
\]

the hydrodynamical equation (equation A.43) is shown to be equivalent to Burgers’ equation

\[
\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}. \tag{A.45}
\]
Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Journal/Publication Details</th>
</tr>
</thead>
</table>