DISSERTATION

LARGE-SCALE COMPUTATIONAL ANALYSIS OF NATIONAL ANIMAL IDENTIFICATION SYSTEM MOCK DATA, INCLUDING TRACEBACK AND TRACE FORWARD

Submitted by

Joshua Ladd

Department of Mathematics

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2008

COLORADO STATE UNIVERSITY

November 5, 2008

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY JOSHUA LADD ENTITLED "LARGE-SCALE COMPUTATIONAL ANALYSIS OF NATIONAL ANIMAL IDENTIFICATION SYSTEM MOCK DATA, INCLUDING TRACEBACK AND TRACE FORWARD" BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Dr. David Zachmann

Dr. James Thomas

Mr. John Picanso

Dr. John Scanga

Adviser: Dr. Patrick J. Burns

Department Head: Dr. Gerhard Dangelmayr

ABSTRACT OF DISSERTATION

LARGE-SCALE COMPUTATIONAL ANALYSIS OF NATIONAL ANIMAL IDENTIFICATION SYSTEM MOCK DATA, INCLUDING TRACEBACK AND TRACE FORWARD

Cattle production is the single largest segment of U.S. agriculture. Animal disease, whether a single incident or a full-scale outbreak, can result in significantly restricted access to both foreign and domestic markets. Regaining consumer confidence is difficult. If a disease cannot be traced back to a common source, then only time can tell whether or not eradication and containment efforts have been successful. Simply "waiting it out" can result in long-term economic losses on a National scale especially when diseases which are prone to epizootic outbreaks or those with long incubation periods are involved.

The United States Department of Agriculture (USDA) maintains that traceability is the key to protecting animal health and marketability. The National Animal Identification System (NAIS) is a voluntary disease traceability framework released by the USDA. Many of the efforts surrounding the development of the NAIS have encompased the identification of livestock production and handling premises as well as individuals or herds of animals, whereas little effort has been directed toward the ultimate goal of animal traceback in 48 hours.

In this dissertation, computational science is applied to the problem of animal disease traceability. In particular, a computational model is developed for the purpose of conducting large-scale traceability simulations. The model consists of two components; the first being a parallel, Monte Carlo discrete events simulator capable of generating large, NAIS-compliant, mock datasets representative of the processing requirements of actual NAIS data. The second component is a large-scale, parallel disease tracing algorithm that is mapped onto an SMP supercomputer where high-performance is achieved by adopting a hybrid parallel programming model that mixes a shared memory multi-threading model (OpenMP) with a distributed memory message passing model (MPI).

The objectives of this dissertation are to characterize the computational requirements of the NAIS, identify computational platforms and programming paradigms well suited to this effort, and to identify and address computational performance bottlenecks associated with large-scale tracing algorithms.

Joshua Ladd
Department of Mathematics
Colorado State University
Fort Collins, Colorado 80523
Fall 2008

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my PhD advisor and mentor, Dr. Patrick J. Burns, without whom none of this would have been possible. Your patient advising, expertise, wisdom and seemingly endless knowledge have slowly and meticulously sculpted a raw analytical intellect into a confident and independent computational scientist. I am forever indebted to you, and I will never forget the immeasurable impact you had on both my scientific and intellectual development. A good PhD project is nothing without a great mentor, thank you Pat.

I would like to express my gratitude to my PhD committee members Dr. Dave Zachmann, Dr. James Thomas, Dr. John Scanga, and Mr. John Picanso. The time and effort you spent refereeing this research is greatly appreciated. Your valuable comments have added significantly to the scientific quality of this work.

I would also like to express my gratitude to Dr. Dan Pryor of the Supercomputing Research Center for the many early morning conference calls we had which were always intellectually productive and just a lot of fun in general. Your expertise is highly valued and lent much insight during our research meetings.

I would like to express my thanks to the National Energy Research Scientific Computing Center (NERSC) and the Lawrence Berkeley National Lab for providing me with the state-of-the-art supercomputing resources

necessary to carry out this research. I cannot say enough good things about NERSC and LBNL in general. This truly is a world-class scientific computing facility in every respect.

I would like to acknowledge two very important mentors who played significant roles in my early career; Mr. Brian Jones for introducing me to the beauty of physics and encouraging me to pursue a career in cosmology, and Dr. Ken Klopfenstein who "discovered" my mathematical talent and provided me with an analytical foundation upon which my entire scientific career has been built.

Finally, I would like to thank my family and, in particular, my parents who have stood by me and given their unconditional support from day one. Words cannot express how grateful I am for all of your love and support. Without it, I surely would not have made it through this ordeal and I am so happy to share this accomplishment with both of you.

DEDICATION

For my parents

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Domestic Impact of Animal Disease

Cattle production is the single largest segment of U.S. agriculture. With a herd of over 100 million animals [68], and growing, the value of beef and dairy surpassed $31 billion ($19.4 and $12.3 billion, respectively), or about 40% of total U.S. agricultural production in 1999 [67]. At the consumer retail level, beef sales have started to rebound from a steady 20-year decline that has seen per capita beef consumption cut in half from its 1980 levels. In 1999, consumer sales posted a record $52 billion, with mean yearly consumption of 69.6 pounds per person [67]. Exports also represent a sizable portion of U.S. agricultural output. For example, the United States sold $2.7 billion in beef to trading partners in 1999. Four countries currently buy 95% of U.S. beef exports. Japan is the principal buyer ($1.4 billion), followed by Mexico ($454 million), Korea ($331 million), and Canada ($273 million) [67].

Access to foreign and domestic markets can be severly restricted or prohibited altogether in the event of an animal disease outbreak. Such was the case in 2003 when the first reported instance of Bovine Spongeform Encephalopathy (BSE) (also known as *mad cow disease*) in the United States was announced. BSE is not common in the U.S. and feeding regulations

in this country make it unlikely that an outbreak in the U.S. would be widespread, however the following case study illustrates how a high profile disease can have a serious impact on the confidence of consumers and trading partners.

BSE is a chronic, degenerative disease affecting the central nervous system of cattle. BSE was discovered in Britain in 1986 and has remained a worldwide concern to the present day. BSE spreads among cattle primarily through feed containing meat and bone meal made from rendered ruminant products of infected animals [68].

The first diagnosis of BSE in the U.S. occurred on December 23, 2003. A second case was discovered in 2005 in a twelve-year old cow in Texas. A third case was discovered in 2006 in a ten-year old cow in Alabama. A large number of cows associated with the index herd were untraceable in each investigation. Each BSE case required the investigation of at least eight different herds and the three investigations took more than 155 days to complete [68, 67].

The short-term economic impact was devastating. U.S. beef exports dropped from a record 2.5 billion pounds in 2003 to 461 million pounds in 2004, a fall of over 80%. The outbreak cost the beef industry over $2 billion in 2004 alone [68, 67].

In the U.S., a large-scale BSE outbreak is unlikely, however animal diseases prone to epizootic outbreaks such as foot-and-mouth disease (FMD), bovine tuberculosis and exotic Newcastle disease (a viral infection of birds) remain clear and present dangers to U.S. agriculture. Indeed, the financial impact of recent animal disease investigations highlight this fact [67].

Bovine Tuberculosis

- Since 2002, detections in Arizona, California, Michigan, Minnesota, New Mexico and Texas have required the destruction of more than 25,000 cattle. A new detection in June in New Mexico will add to this total.
- USDA has spent approximately $130 million dollars on owner idemnification and control activities.
- Producers are financially affected by strict movement controls applied after new detections.
- Since 2004, USDA has tested 787,000 animals in response to TB outbreaks.

Exotic Newcastle Disease (2002)

- Confirmed in California and quickly spread to the neighboring states of Arizona, Nevada, and Texas.
- Largest animal disease outbreak in the United States in 30 years. It took 10 months to eradicate the disease at a cost of $180 million.
- Poultry producers, both commercial and backyard flock owners, lost 4 million birds during extensive depopulation activities.

Bovine Spongeform Encephalopathy (2003)

- USDA spent $5 million on its epidemiology investigation, depopulation, and initial response.

- The United States lost 80 percent of its foreign beef trade.
- As part of the effort to regain access to foreign markets, the USDA spent approximately $189 million on the enhanced BSE surveillance program.

1.2 Impacts Elsewhere

A comparison with the European livestock industry experience shows one possible scenario for what is at stake. In the EU, the beef industry suffered two major losses from their BSE crisis: (1) a 20-30% decline in domestic beef sales due to negative long-term effects on consumers' confidence, and (2) losses in international trade in cattle, beef, and feed.

Since the BSE epidemic began in Great Britain in 1986, Europe's cattle and meat industries have undergone a significant increase in regulation. Animal protein feed bans, quarantines, surveillance, increased testing, herd renewal, and selective cull measures are now in effect in many EU nations. In Great Britain, where the BSE epidemic has reached 179,000 confirmed cases in cattle since 1986, these measures appear to be resulting in a steady decrease in the number of infected cattle from the 1992 peak. One program, put in action following the 1996 U.K. beef and cattle ban by the EU, is the so called "Over Thirty Month Slaughter" scheme (OTMS). Under this plan, which bans the sale of meat from cattle aged over 30 months old, the U.K. has destroyed over 4.5 million animals, at a cost of $4 billion. Similar EU programs (which include feed bans, mandatory animal testing and tracing, and OTMS) could go into effect in Germany, Italy and Spain. Germany, for instance, expects to destroy about 400,000 cattle under a "purchase for destruction" program [54].

In the United States, after the first BSE oubreak and subsequent economic devastation, it is now widely believed that the implementation of a National animal traceability system that leverages current and evolving information technology is vital to safegaurding the U.S. food supply, protecting human health, and mitigating economic risk [54].

1.3 Mitigating Risk

The United States Department of Agriculture maintains that traceability is the key to protecting animal health and marketability [67]. In order to respond quickly and effectively to an animal disease event (whether it is a single incident or a full-scale outbreak), animal health officials need to know which animals are involved, where they are located, and what other animals might have been exposed. The sooner reliable data is available, affected animals can be located, appropriate response measures can be established, and disease spread can be halted.

Retrieving animal locations and movement data within 48-hours is optimal for efficient, effective disease containment [67]. As evidenced from case studies, current U.S. animal disease traceablity infrastructure falls well short of this objective . The U.S. Department of Agriculture (USDA) is focusing on opportunities to bolster disease tracing capabilities by increasing the quantity and quality of animal identification data and the efficient use of evolving technology solutions.

1.4 Leveraging Technology to Enhance Traceability

In April 2006 the USDA-APHIS released a voluntary animal identification and traceability framework collectively known as the National Animal Identification System (NAIS). To date, much of the effort surrounding

the development of the NAIS has encompassed the identification of livestock production and handling premises, the identification of individuals or heards of animals, and harmonizing government and industry animal identification programs by standardizing data elements of disease programs to ensure compatability. However, little effort has been directed towards the accomplishment of the ultimate goal of the program which is an animal traceback in 48 hours [52]. Typically, epidemiological investigations are conducted through manual record retrieval and review. While increasing the quality and quantity of data clearly increases traceability, it can also present a significant barrier to the investigation as data acquisition efforts have far outstripped data analysis efforts.

This dissertation will apply computational science to the problem of animal disease traceback. For a given volume of movement data, it is not known what the computational processing requirements are for achieving rapid traceback. In this dissertation, we develop a computational model of animal disease traceability for the purpose of conducting large-scale traceability simulations. The goal of this approach is to characterize the computational requirements of NAIS dataset processing.

1.5 Overview of Dissertation

In Chapter two, a computational architecture is presented. It is argued, via empirical benchmarking, that the platform is a good target to map a high-performance disease tracing algorithm onto. In Chapter three, a new, fast, portable parallel random number generator is implemented and the parallel performance and accuracy is assessed. The results of Chapter three are employed in the implementation of a parallel Monte Carlo discrete

events simulator which is described in Chapter four. Chapter four details a Monte Carlo process that models large animal populations for the purpose of conduction large-scale traceability simulations. The mock datasets generated consist of NAIS-compliant data and range in size from 2 million up to 100 million animals, representative of a "National" dataset. In Chapter five, a large-scale, parallel disease tracing algorithm is mapped onto the target architecture. High-performance is achieved by adopting a hybrid parallel programming model that mixes OpenMP with MPI which facilitates efficient use of and access to system memory. The datasets created in Chapter four are processed and computational performance is measured as problem size is scaled-up to that of the "National" dataset. Finally, in Chapter six, a summary of the major conclusions of this work are presented along with recommendations for further work.

Chapter 2

MEASURED PERFORMANCE OF THE TARGET ARCHITECTURE

A major distinction between good sequential algorithm design and good parallel algorithm design is that the latter is typically much more intimately dependent upon the target architecture. Since parallel algorithms are generally designed to optimize communication overhead, platform dependent variables, e.g. interprocessor stream rates, induce a topology on an algorithm's performance space. In particular, an algorithm's *scalability* and *efficiency* are complicated functions of the architecture onto which the algorithm is mapped, requiring particular knowledge of the system characteristics.

Indeed, hardware characteristics that can affect an algorithm's parallel performance include; processor frequency and memory performance, heterogeneous memory paths, interprocessor communication latencies and stream rates, shared memory performance in a multi-core environment, and, at times, I/O performance. Making optimal use of a large-scale architecture requires quantitative insights gained through empirical benchmarking and performance profiling. The measured performance can be used to inform how best to map an algorithm to a specific target architecture. In the remainder of this chapter, the performance space of the target architecture

is mapped by measuring processor and memory performance, shared memory performance, distributed memory performance, and I/O performance as both problem size and the number of processors are scaled-up.

2.1 The Bassi System

The research presented in this dissertation used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. In particular, this research was awarded multiple allocations at NERSC that provided access to state-of-the-art supercomputing architectures.

The target architecture is the NERSC *Bassi* system, an IBM p575 POWER5 distributed memory computer with 512 processors available to run scientific computing applications. The processors are distributed among 64 compute nodes, each of which contains 8 processors. A compute node is an 8-way symmetric multiprocessor (SMP). Processors within each node have a shared memory pool of 32 GBytes. Each POWER5 processor core has a theoretical peak performance of 7.6 GFlops or 3.89 TFlops theoretical peak system performance. The compute nodes are interconnected with the IBM *Federation* HPS switch, a high-bandwidth switching network which is connected to a two-link network adapter on each node. Each node runs its own full instance of the standard AIX operating system. Detailed system specifications are provided below [57]:

•IBM Cluster

- 8-way single SMP cores per node

- 1.9 GHz single-core POWER 5 64-bit processors (DCM: Dual Chip Module with one active core)
- 7.6 GFlops theoretical peak per processor
- 64 KBytes 2-way associative Instruction cache
- 32 KBytes 4-way associative Data cache
- 2 MBytes on-chip L2 (10-way associative, 3x640 KBytes)
- 36 MBytes L3 cache, with a L3 to L2 peak bandwidth of 243.2 GBytes per second.
- 200 GBytes/sec cumulative peak theoretical memory bandwidth
- 32 GBytes memory per node
- 48 GBytes/sec theoretical peak I/O Bandwidth

•Production System

- 64 Compute nodes (512 compute processors)
- 64*8*7.6 GFlops = 3.89 TFlops theoretical system peak performance

•I/O Subsystem Configuration

- 6 Virtual Shared Disk servers to support GPFS, each with 2 High Performance Switch (HPS) links. (See next section)
- Each Virtual Shared Disk server has sixteen 2 Gbps Fibre Channel links

•High Performance Switch (HPS)

- Each node: One 2-link High Performance Switch adapter
- Each node: attaches to the interconnect with 2 links, one to each of 2 separate planes
- LAPI and MPI communication
- Data uses LAPI over HPS
- Peak HPS bandwidth - 2 GBytes per second per link each direction
- MPI latency: less than 5 μs

•Relevant Software

- IBM AIX 5.3
- IBM Parallel Environnent 4.2
- IBM C/C++ Enterprise Edition 7.0
- IBM Fortran Enterprise Edition 9.1

2.2 Measured Performance

In this section, various algorithms are used to call forth performance characteristics of the architecture as the problem is scaled-up. All results shown are averaged over 100 trials so as to produce repeatable, average values of execution rates.

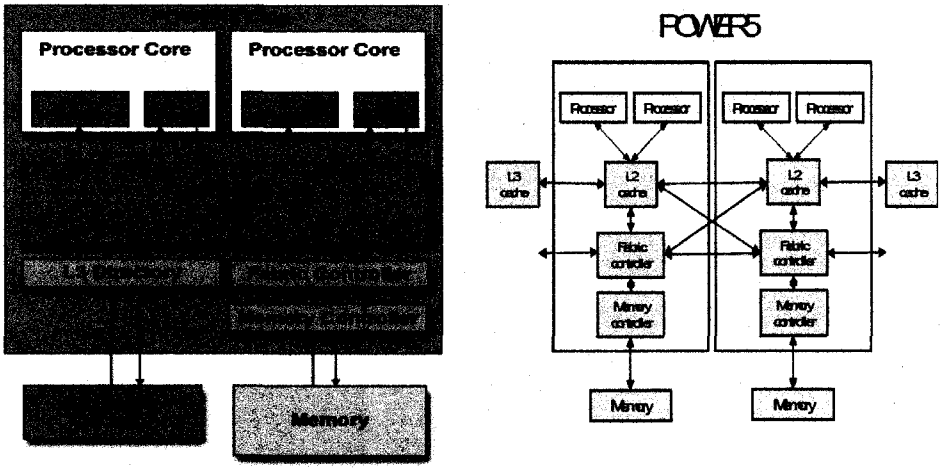

Figure 2.1: High-level system structure for a dual-core IBM POWER5 system is shown on the left [57]. On the right, two POWER5 chips are interconnected [43]. On Bassi, only a single processor core is active, eight single-core POWER5 chips make up a single 8-way SMP node.

2.2.1 Processor Performance

The IBM POWER5 chip used on the Bassi system is a single-core chip configured on a dual chip module. The dual chip module is depicted in Figure 2.1. Each chip and hence each processor core has its own cache hierarchy consisting of an on-chip L1 and L2 cache and an off-chip L3 cache which serves as an L2 victim cache. The objective of this section is to measure the peak achievable floating point operation performance of a single processor core. Peak achievable performance is not simply a function of processor frequency, 1.9 GHz in this case, performance is a complicated function of factors such as the number of functional units, pipeline depth, translation lookaside buffer (TLB) size, cache performance, and memory access patterns.

Dense matrix multiplication is the widely accepted floating point operation benchmarking kernel [21]. This linear algebra operation is a paradigm

in the study of performance optimization of numerically intensive codes. The peak achievable performance of a single processor core will be measured by multiplying two dense, square matrices together with the highly optimized Fortran 90 intrinsic function, *matmul*, which has been tuned to this particular architecture.

Let A, B be two $N \times N$ matrices. Denote the i, j^{th} component of the product AB by $[AB]_{i,j}$. Then

$$[AB]_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}. \tag{2.1}$$

Each component of the product, AB, requires $2N - 1$ Flops to compute, thus dense matrix multiplication has cubic complexity.

In Figure 2.2, on the left, the average execution time as a function of matrix size is depicted (red-squares) along with a least-squares fit of the data to a cubic polynomial (blue-line). On the right, the average floating point execution operation rate, measured in MFlops, is plotted as a function of matrix size. The data suggests that an individual processor core is capable of achieving nearly 36% of its theoretical peak on a non-trivial application that requires significant low-level optimization in order to achieve high performance.

Each Flop requires two loads and a single store. Hence taking 2.7 GFlops as peak achievable floating point operation performance, the local memory path has a peak achievable bandwidth of 65 GBytes/sec.

2.2.2 Memory Performance

System memory and cache performance are measured by computing the dot product of two large arrays at various strides. Computationally, this is accomplished by executing Algorithm 1 for various values of S and

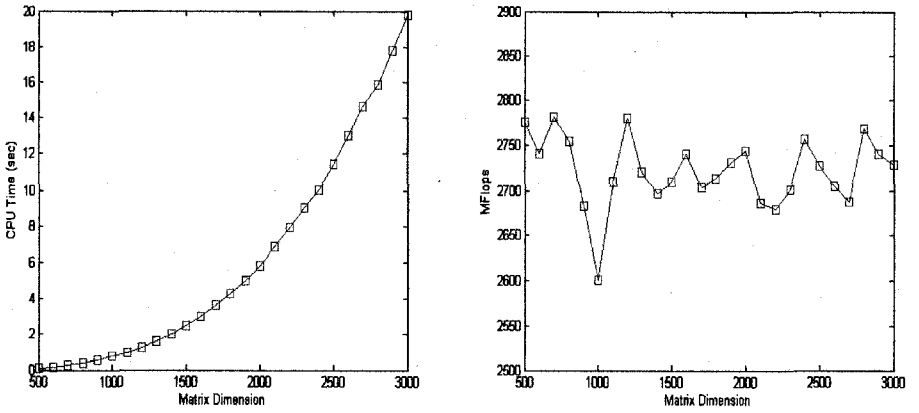

Figure 2.2: Performance results for dense matrix multiplication at various scales. On the left, the average exectution time versus matrix size is plotted (red-squares), a least-squares approximation, plotted as a solid blue line, fits the data to a cubic polynomial. On the right, average processor performance, measured in MFlops, is plotted as a function of matrix size.

measuring the execution time. In each experimental trial, N is fixed at 90 million and each array contains N, 8-byte, double-precision words. Longer strides will increase the cache miss rate which should decrease processor performance. The POWER5 cache hierarchy, characterized on page 5 and seen schematically in Figure 2.1, is similar in design to the cache systems described and analyzed in [69, 24].

Algorithm 1 Computes the dot product with stride S.

1: double $A(N)$, $B(N)$
2: **for** $(i = 1 : S : N)$ **do**
3: $\quad sum \leftarrow sum + A(i) * B(i)$
4: **end for**

In Figure 2.3 the processor performance, measured in MFlops, is plotted versus stride. Performance degrades as $O(S^{-1})$, as expected, and levels-off for strides larger than 500 double words. This empirically determines cache-line size to be 500 doublewords on this machine and the time to

transfer a cache line from memory to cache is about 0.157 μs. Thus spatial locality is maximized when data and access patterns are structured so that strides do not exceed 500 double-precision words.

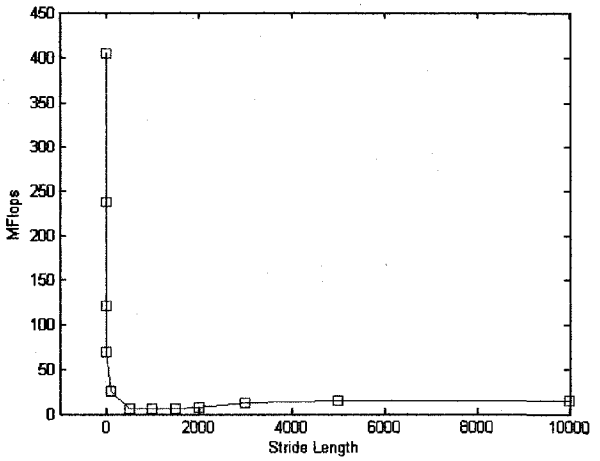

Figure 2.3: Processor performance, measured in MFlops, is plotted as a function of stride. The plot was generated by computing the dot product of two large arrays at various strides.

In this experiment, processor performance is quite different from that observed for dense matrix multiplication. With unit stride, a single processor core achieves only about 1.5% of its peak theoretical performance. Two observations explain this disparity. The first is clear, in the dot product experiment each array contains 90 million 8-byte, double-precision words. Whereas the largest matrix studied contains only 9 million 8-byte, double-precision words. In comparison, the memory requirement of the dot product experiment far exceeds that of the matrix multiply experiment resulting in many more off-chip memory calls and, as expected, leads to an overall performance degradation. When the length of the arrays in the dot product experiment are decreased, overall performance improves. A second and more subtle reason for the observed performance disparity lies in the fact

that matrix multiplication is accomplished with a Fortran 90 intrinsic and the dot product is computed with unoptimized code. On this platform, when the intrinsic *matmul* is called, a high-performance implementation, tuned to this particular platform, is invoked resulting in significantly better floating point performance.

2.2.3 Shared Memory Performance

Each SMP node consists of eight single-core POWER5 chips which share 32 GBytes of memory. The objective of this benchmark is to quantify the cost to access shared memory as the number of processor cores within a node increase. Algorithm 2 reads an array, A, with stride, S, 100 times. Reads are processed concurrently by a team of OpenMP threads which share the array A. After each read, the data or fractions of it, are in cache for the next read. The cache is flushed after each read in order to equalize the cost of successive reads. Performance is assessed by measuring the parallel speedup as the number of threads increases.

Algorithm 2 Multi-threaded reads with stride S.

```
1: double A(N), B(100)
2: !$OMP PARALLEL DO
3: for (i = 1 : 100) do
4:    flush cache
5:    for (j = 1 : S : N) do
6:       B(i) ← A(j)
7:    end for
8: end for
9: !$OMP END PARALLEL DO
```

Results in Figure 2.4 are separated into two categories; small-stride (left) and large-stride (right). For small strides, the parallel performance is

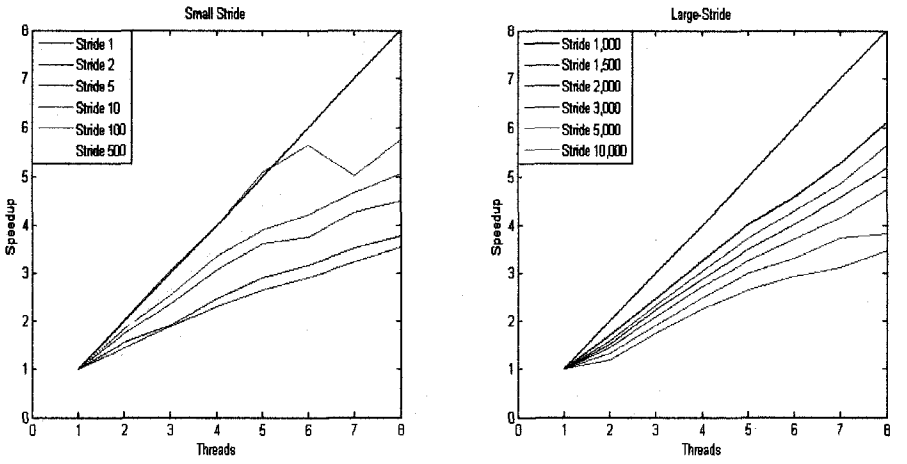

Figure 2.4: On the left, small-stride shared memory parallel performance within a node. On the right, large stride performance (right). Bandwidth contention degrades shared memory performance as the stride is increased.

observed to increase with stride length. Performance is maximal for strides in the range of 100 - 500 double words. For large strides, parallel performance monotonically decreases with increased stride and slowly approaches a steady-state speedup.

Although Algorithm 2 is embarassingly parallel, the performance in Figure 2.4 is rarely observed to be optimal. This is common in shared memory systems since usually the original memory path is shared by multiple processors. Degradation occurs when processors compete for limited shared memory bandwidth. The experiment suggests, however, that shared memory performance can be improved by making efficient use of the cache hierarchy. Since each POWER5 chip has its own cache hierarchy, finding the optimal stride and structuring access patterns accordingly minimizes shared memory traffic and can maximize individual processor performance. This experiment is designed to quantify the single-access cost per processor per stride. Because the cache is flushed after each read, the effects of

temporal locality are unaccounted for in these observations. Reusing data can improve shared memory performance as well by taking full advantage of the large L3 cache which serves as an L2 victim cache (temporal locality is a good way to exploit a victim cache) and resides on the processor side of the fabric controller.

2.2.4 Distributed Memory Communication Performance

On Bassi, distributed memory parallelism is facilitated by the *Message Passing Interface* (MPI) API. On this platform, not all MPI communication goes across the switch. For communications among tasks that reside on the same node, MPI messages are instead routed through fast shared memory buffers. The objective of the next benchmark is to measure MPI transfer rates for inter-nodal and intra-nodal communication. In this context, the transfer rate is measured by timing how long it takes to send a one-way message of a given size. Timings account for both latency and stream rate. Both one-to-all collective messaging (MPI broadcast) and point-to-point (MPI send/receive) messaging are considered.

Comparing timings and usage of global collectives is much simpler than surveying the space of possible pairwise communications. The performance of point-to-point messaging depends closely on the pattern of messages in a code. In this experiment, the point-to-point transfer rate is measured by sending a one-way message of a given size to a neighboring MPI task. In order to test one-to-all messaging, a message of a given size is broadcast to all active processors. In both experiments, the effect of bandwidth contention on transfer rates is assessed by increasing the number of active processors. In the point-to-point experiment send/receive pairs send messages concurrently.

Results for intra-node messaging are depicted in Figure 2.5. On the left, average transfer rates for send/receive pairs are plotted as a function of message size. Messages in the range of 1 MByte in length (10^5 words) exhibit optimal performance topping out at 6 GBytes/sec when all eight processors are active. On the right, the average transfer rates for an MPI broadcast are plotted as a function of message size. Broadcast transfer rates are comparable to those for point-to-point messaging.

Results for inter-node messaging are depicted in Figure 2.6. On the left, the average transfer rates between a send/receive pair are plotted as a function of message size. Bandwidth peaks for large messages approaching a steady-state transfer rate of about 1.8 GBytes/sec. On the right, the average transfer rates for an MPI broadcast are plotted as a function of message size. When the number of nodes is large, broadcasting across nodes is significantly more expensive than sending many point-to-point messages simultaneously between nodes. Broadcast bandwidth peaks for large messages and approaches a steady-state of about 600 MBytes/sec.

Comparing both possible paths that MPI messages take, the data clearly show that routing small messages through shared memory buffers is much more efficient than routing them through the HPS switch. For large messages, point-to-point transfer rates within a node are comparable to those between nodes. This observation suggests that shared memory buffers have much lower latency than the HPS switch but have comparable stream rates. This argument is further bolstered by the observation that within a node, the broadcast transfer rate is nearly identical to the point-to-point transfer rate. Thus the main difference between the two data paths is the latency which is significantly larger for the HPS switch. The main

conclusion in all of this is MPI performance is optimal for fine-grained access within a node due to low latency and coarse-grain access between and amongst nodes due to high-bandwidth. and is that Broadcast latency scales well within a node, and scales poorly between nodes.

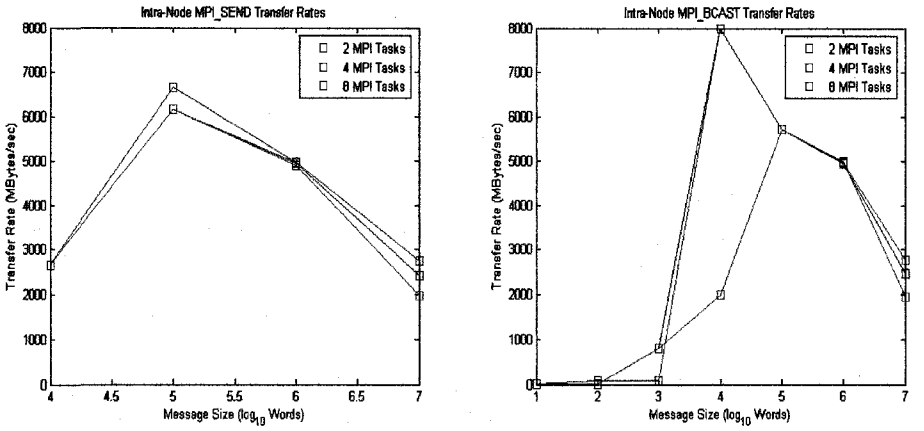

Figure 2.5: On the left, intra-node point-to-point send/receive transfer rate versus message size are plotted. A peak bandwidth of about 6 GBytes/sec can be achieved for messages in the range of 10^5 double-precision words. On the right, the transfer rate versus message size for intra-node one-to-all broadcast are plotted. Performance is comparable or superior to that of point-to-point communication, suggesting that broadcast latency is small within a node.

2.2.5 I/O Performance

The I/O transfer rate is measured by reading a large file into main memory and then writting the same file back to disk. In particular, measurements are made by reading files consisting of 10^3, 10^4, 10^5, and 10^6 double-precision words and subsequently writting them back to disk. On this architecture, the I/O transfer rate is observed to be independent of the physical location of a processor. The average input transfer rate is 4.87 MBytes/sec and the average output transfer rate is 3.8 MBytes/sec.

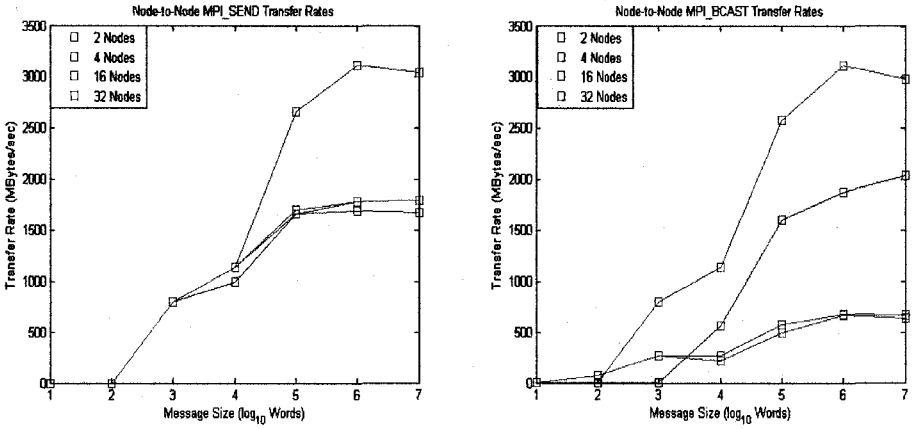

Figure 2.6: On the left, inter-node point-to-point send/receive transfer rate versus message size are plotted. Peak bandwidths are approached as message size increases, as expected for a high bandwidth switch. On the right, the transfer rate versus message size for inter-node one-to-all broadcast are plotted. Broadcast performance suffers due to the high latency associated with barrier synchronization over the HPS switch.

2.3 Conclusions

In conclusion, the NERSC Bassi system is a high performing computational platform with fast, low-latency shared memory SMP nodes which are interconnected over a high-bandwidth switch network. An individual POWER5 processor core has an achievable peak speed of 2.7 GFlops, attaining 36% of its peak theoretical speed.

The POWER5 memory hierarchy consists of a dual-level on-chip cache system and a single off-chip Level 3 cache that can greatly improve floating point performance for in-cache data access. The length of a cache line is empirically determined to be 500 double words and reading a cache line in from memory to cache takes about 0.157 μs. Spatial locality is optimal for data access patterns with strides smaller than or equal to 500 double words.

Shared memory performance scales poorly for both very large and for very small strides. Optimal strides for single use data are in the range of 100-500 double words. Performance can be further improved by making efficient use of the large L3 victim cache by increasing temporal locality.

Distributed memory performance is heterogeneous and depends on the path the data travels. For point-to-point messages sent over the HPS switch, a steady-state transfer rate of 1.8 GBytes/sec is approached as both the number of nodes and problem size is scaled up. Inter-node MPI broadcast transfer rates scale poorly due to latency associated with synchronizing nodes over the HPS switch. Broadcast transfer rates approach 500 MBytes/sec.

Messages between MPI tasks on the same node do not travel over the HPS switch but are instead routed through shared memory buffers. This can result in performance gains of over a factor of three for messages of size 10^7 8-byte double words. Messages in this range have an achievable transfer rate of 6 GBytes/sec. Intra-node latency is small in comparison to the latency of the HPS switch, but stream rates are comparable. MPI performance is optimal for fine-grained access within a node, due to low latency, and coarse-grain access between and amongst nodes because of the high-bandwidth HPS switch.

The average disk input stream rate is measured to be 4.8 MBytes/sec and the average disk output (write) stream rate is found to be 3.8 MBytes/sec.

Table 2.1 suggests that high performance can be achieved by leveraging both the fast, low-latency shared memory within a node and the high-bandwidth HPS interconnect. For a memory intensive application, a hybrid parallel programming model that mixes shared memory multi-threading

	Achievable Bandwidth (GBytes/sec)	Relative Cost
Local memory	65	1 : 1
Intra-node send/receive	6	11 : 1
Intra-node broadcast	6	11 : 1
Inter-node send/receive	1.8	36 : 1
Inter-node broadcast	0.5	130 : 1
Input	0.0048	13500 : 1
Output	0.0038	17100 : 1

Table 2.1: Relative cost of memory access.

within a node and message passing between nodes, to the extent possible, is the most effective strategy to maximize parallel performance. Even though shared memory contention can lead to some performance degradation.

Chapter 3

A FAST, PORTABLE, PARALLEL RANDOM NUMBER GENERATOR

At large-scale, the performance and accuracy of a Monte Carlo application can depend in a highly nontrivial way on the parallel random number generator (RNG) employed. Monte Carlo methods involve the deliberate use of random numbers in a calculation that has the structure of a *stochastic process.* A stochastic process is a sequence of states whose evolution is determined by random events that, on a computer, are determined by pseudo-random numbers [29] (hereinafter, we will omit the 'pseudo', consistent with standard paractice). Monte Carlo calculations have in the past, and continue to, consume a significant fraction of available high-performance computing cycles [60]. This is due, in part, to the fact that some important Monte Carlo calculations lend themselves to a highly efficient and portable parallelization.

On a computer, an RNG is actually a deterministic algorithm that typically produces a periodic sequence of states by means of a linear recursion that appears random to an application. *Effective* RNGs of this type have extremely long periods so that an application is never affected by this periodic structure. In addition to possessing long-periods, an effective *sequential* RNG must be free of intra-stream correlations that can bias the outcome

of a stochastic process. *Parallel* RNGs must further provide an algorithm which allocates the state-space of the RNG to different processors.

It is necessary to subject any RNG to a rigorous comprehensive analysis and assessment before it is deployed in a large-scale application. Current large-scale Monte Carlo computations may consume the entire periods of many older generators in only a few seconds [60]. Tests on important applications at large-scale have revealed defects in RNGs that were not apparent when run on smaller simulations [18, 22]. A defective RNG can insidiously compromise the accuracy of a Monte Carlo calculation at large-scale and is difficult, if not impossible, to diagnose at runtime.

The most effective manner in which to assess the quality of an RNG is by direct empirical testing of the streams produced. While the quality of an RNG sequence is extremely important, the unfortunate fact is that little mathematical theory exists to assess the quality of the current, most sophisticated generators. Though some theoretical results exist in the literature, most of the theory is limited to defining criteria for achieving a maximal period since mathematical bounds on correlations are extremely difficult to prove. The situation is further complicated for parallel RNGs by the fact that effective parallel RNGs must be free of both intra-stream correlations within individual streams as well as inter-stream correlations between and among streams on separate processors.

Empirical testing falls into two broadly defined categories: (i) statistical tests, and (ii) application-based tests. Statistical tests compare some statistic obtained with an RNG sequence to the expected statistic assuming the sequence were truly random. Applications-based tests employ an RNG in an actual application with a known solution or convergence rate. If the

results of any of the tests are sufficiently far from those expected, then the RNG is considered suspect. If the outcome of most of the tests are far from the expected, then the RNG is considered defective. In some cases, statistical tests can be used to locate isolated defects which may be treated and cured. This type of *tuning* can play an important role in long-term code maintenence.

This chapter presents a novel enhancement and implementation of a fast, portable parallel random number generator. The implementation is assessed through rigorous empirical testing. Both types of empirical testing, statistical and applications-based, are employed. In particular, a subset of Knuth's so-called stringent tests and Marsaglia's Diehard tests, which have been particularly effective at exposing defects in parallel RNGs, are used. The proposed parallelization is implemented in a Monte Carlo photon heat transfer code which has a known solution and convergence rate. As an application-based test, the heat transfer code is run at large-scale and the empirical outcome is compared to what is theoretically expected.

3.1 Additive Lagged Fibonacci Generators

The definitive reference on sequential RNGs is Knuth [31]. Some implementations of sequential algorithms can also be found in *Numerical Recipes* [46]. Perhaps the most accessible exposition is that of Anderson [3], who presents some excellent illustrative graphics. Of particular interest to the present effort is the family of linear recurssions collectively termed, Additive Lagged Fibonacci Generators (ALFG). This type of sequential generator has been extensively tested for randomness [38] and given high marks.

To begin, consider the class of additive lagged fibonnacci generators which are defined by the family of linear recursions

$$x_n = (x_{n-l} + x_{n-k}) mod(M), \quad l > k > 0. \tag{3.1}$$

In order to compute the $(l+1)^{th}$ value, the l previous values are required. Let the l most current values occupy the $l \times 1$ state-vector

$$\vec{r} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{l-k} \\ \vdots \\ x_{l-2} \\ x_{l-1} \end{pmatrix}. \tag{3.2}$$

Then equation (3.1) defines a linear transformation, $T : \mathbb{R}^l \rightarrow \mathbb{R}^l$, which acts on the state-vector in (3.2) as

$$T \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{l-k} \\ \vdots \\ x_{l-2} \\ x_{l-1} \end{pmatrix} \rightarrow \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{l-k} \\ \vdots \\ x_{l-2} \\ x_{l-1} \\ x_l = x_0 + x_{l-k} \end{pmatrix}. \tag{3.3}$$

The state vector (3.2) is called the *ALFG register* and contains the l most current values which are required to compute the next value in the sequence. We say that l is the length of the ALFG register and that k is the *tap point.* For computational efficiency the value of the modulus M is usually chosen to be a power of 2, i.e. $M = 2^m$ where for a 32-bit signed integer it makes sense to choose $m = 31$. The structure of the ALFG in (3.1) is completely characterized by the register length, tap point and modulus, thus we adopt

the notation $LFG(l, k, m)$ where m is the power of two used for the modulus M. The *initial seed* is defined as an initial fill of the ALFG's register with l words and is denoted $\vec{r}_0$. The first l words must be filled by an application; the process in which the initial seed is assigned is termed *initialization*. In this work, initialization is done at the bit level by employing FORTRAN bit-wise intrinsic functions. With this in mind, the initial seed is viewed as an $m \times l$ binary matrix with $2^{ml} - 1$ nontrivial bit-wise states. Table

b_{m-1}	*	*	*	*	*	*	*	*	*	MSB
b_{m-2}	*	*	*	*	*	*	*	*	*	
$\vdots$	*	*	$\ddots$	*	$\ddots$	*	$\ddots$	*	*	
b_1	*	*	*	*	*	*	*	*	*	
b_0	*	*	*	*	*	*	*	*	*	LSB
	x_0	x_1	$\cdots$	x_c	$\cdots$	x_{k-l}	$\cdots$	x_{l-2}	x_{l-1}	Word

Table 3.1: Binary matrix perspective of an ALFG initial seed.

3.1 depicts an ALFG register as a binary state matrix. Each word in the register is an $m - bit$ integer where the most significant bits (MSB) of the register occupy the top row of the matrix and the least significant bits (LSB) occupy the bottom row.

3.1.1 Period of the ALFG

For the ALFG with power of two modulus, the maximum possible period is huge: $(2^l - 1)2^{m-1}$ [39]. However, notice that the maximum possible period is considerably smaller than the number of nonzero fills, $2^{ml} - 1$. The missing state space can be accounted for by the large number of disjoint maximum period cycles. A necessary condition for achieving a maximal period is that the characteristic polynomial associated with the recurrence, $f(x) = x^k + x^l + 1$ be primitive modulo 2, $l > 2$ [8]. The only additional requirement to obtain a full period cycle is that not all least

significant bits be zero. In terms of residues modulo 2, this is achieved if one or more words of the initial seed $\vec{r}_0$ is odd. A simple computation shows that the number of initial seeds that give maximal possible period is $(2^l - 1)2^{l(m-1)}$. Since each of these initial seeds is in a maximum possible period cycle, there must be

$$E = \frac{(2^l - 1)2^{l(m-1)}}{(2^l - 1)2^{m-1}} = 2^{(l-1)(m-1)} \tag{3.4}$$

cycles with maximum possible period. Following [39], if we define an equivalence relation among initial seeds so that two seeds are equivalent if they are in the same cycle, then the generator has E distinct maximal period cycles.

3.1.2 A Canonical Form

The state space of the ALFG as described above is toroidal [40] with equation (3.1) providing the algorithm for movement in one torus dimension. Mascagni *et al.* [39] have provided an elegant algorithm for movement in the second torus dimension (movement from cycle to cycle) by exploiting the following result.

Theorem 3.1.1. *Suppose l and k are such that $x^l + x^k + 1$ is primitive modulo 2. Then for every m – bit initial seed $\vec{r}_0$ which has at least one odd element, there exists an integer $0 \leq p \leq (2^l - 1)2^{m-1}$ such that the register state, $T^p\vec{r}_0$ has the form shown in Table 3.2. That is, the first word is 0 and all words except for word x_c are even. Furthermore, the index of the single odd word is uniquely determined by l and k.*

Theorem 3.1.1 implies the existence of a canonical form for initial seeds which satisfy the hypothesis. The $(m-1) \times (l-1)$ subarray in Table 3.2

b_{m-1}	0	*	*	*	*	*	*	*	*	MSB
b_{m-2}	0	*	*	*	*	*	*	*	*	
$\vdots$	0	*	$\ddots$	*	$\ddots$	*	$\ddots$	*	*	
b_1	0	*	*	*	*	*	*	*	*	
b_0	0	0	0	1	0	0	0	0	0	LSB
	x_0	x_1	$\cdots$	x_c	$\cdots$	x_{k-l}	$\cdots$	x_{l-2}	x_{l-1}	Word

Table 3.2: The canonical register state of an initial seed which corresponds to a maximal period cycle. The state-space of the $(m-1)\times(l-1)$ canonical rectangle is in one-to-one correspondence with the set of maximal period cycles. The canonical least significant bit position, x_c, is uniquely determined by l and k.

has $2^{(l-1)\times(m-1)}$ possible states. Each one of these states corresponds to a different maximal period cycle. This subarray is appropriately called the *canonical rectangle* and allows us to fully enumerate the set of maximal period cycles. In light of Theorem 3.1.1, the equivalence class previously defined can be restated more precisely; two initial seeds are equivalent if, when put into canonical form, they have the same canonical rectangle state.

3.2 Parallel Implementation

Suppose now that one ALFG sequence is initialized on each of *nproc* processors in a parallel program. A good parallel initialization must satisfy the following criteria [9]:

1. Each processor is assigned a sequence which is taken from a different cycle, easy to accomplish using the canonical form.

2. The initial states of the i^{th} and $(i-1)^{th}$ sequence should look relatively random so as to eliminate short-range inter-stream correlations.

3. A sequence initialized on processor i should be uncorrelated (over the long-range) with a sequence initialized on processor j.

A good parallel initialization can be achieved if, on each processor, the canonical rectangle is bitwise filled with a random bit generator. In particular, the canonical rectangle can be filled as follows:

1. A global initial seed, S_G, is selected by the user and broadcast to all processors.

2. Each processor initializes a *binary shift register* with the binary number $S_G + myid$, where $myid$ is the unique task identifier assigned to each processor.

3. The binary shift register is stepped R times, the output bits placed sequentially in the canonical rectangle.

Provided an effective binary shift regsiter, this initialization satisfies the criteria stated above. In practical application an appropriate choice of the global seed is

$$S_G = (sec + 60 \times (min + 60 \times (hr + 24 \times (day + 30 \times (mon + 12 \times yr))))).$$

Successive runs will therefore generate different sequences, as long as the jobs are launched more than $N/64$ seconds apart. This S_G is guaranteed to fit into 31-bits, is constantly changing, and is unlikely to be identical to a number chosen diliberately by the user.

3.3 Parallel Initialization Schemes

In the proposed parallel initialization, each ALFG stream requires a random bitwise fill of its canonical rectangle. In this section, an effective random bitwise initialization is presented. In particular, two binary shift register schemes are introduced, implemented and their computational efficiencies compared.

3.3.1 Binary Shift Registers

A *binary shift register* (BSR), also known as a *Tauseworthe* generator, is an important class of random number generators. They are particularly well suited to the task of generating random binary bit sequences and, as such, this is their primary usage in modern applications. To understand the mathematical principles underlying a binary shift register, consider the following

Definition 3.3.1. *An n-bit register* $\vec{r}$ *is an* $(n \times 1)$ *state vector whose components are in* $GF(2) = \{0, 1\}$

$$\vec{r} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}. \tag{3.5}$$

In the case of $n = 32$, each state of a 32-bit register naturally coincides with a 32-bit integer by associating b_0 with the most significant bit (MSB) and b_{31} with the least significant bit (LSB) of the integer. For example, the standard unit vectors, $\vec{e}_i$, which have a 1 in the i^{th} component and are 0 in all other components, correspond to the integers 2^{32}, 2^{31},...,2^2,2^1,1 respectively.

Definition 3.3.2. *A binary shift register is a linear transformation recursively applied to an n-bit register which, by successive iteration, produces a sequence of register states*

$$\vec{r}_{k+1} = A\vec{r}_k. \tag{3.6}$$

Where the matrix of the recursion, A, is an element of $GF(2)^{n\times n}$ and addition is modulo 2.

3.3.2 Fibonnaci Register

To begin the discussion of a Fibonacci register (not to be confused with an ALFG), we start with a primitive, irreducible, n-degree polynomial, $p(x)$ with coefficients in the finite field $GF(2)$

$$p(x) = 1 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n. \tag{3.7}$$

For the present effort, primitive polynomial will mean the following

Definition 3.3.3. *We say a polynomial $p(x)$ with coefficients in the field $GF(2)$ is primitive to mean that*

$$< x >= (GF(2)[x]/(p(x)))^{\times}. \tag{3.8}$$

In words, $p(x)$ is primitive if x is a cyclic generator of the multiplicative group of nonzero elements of the finite field, $GF(2)[x]/(p(x))$ which is, itself, isomorphic to $S = \{a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1} | a_i \in GF(2)\} - 0$, the set of all non-zero polynomials of degree $(n-1)$ with coefficients in $GF(2)$. The set S has $2^n - 1$ elements, thus $x \in (GF(2)[x]/(p(x)))^{\times}$ has order $2^n - 1$. By construction, x is a root of $p(x)$ over the field $GF(2)[x]/(p(x))$ which implies the following identity

$$x^n = 1 + a_1 x + a_2 x^2 + \cdots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1}, \quad a_i \in GL(2). \tag{3.9}$$

This identity defines the multiplication in the finite field $GF(2)[x]/(p(x))$. The abstract multiplication can be implemeted in a binary shift register as a rule for forward stepping. Given the most current n values, which are stored in an n-bit shift register, r_n can be computed for all subsequent times as

$$r_n = r_0 + a_1 r_1 + a_2 r_2 + \cdots + a_{n-2} r_{n-2} + a_{n-1} r_{n-1}, \quad a_i \in GL(2). \tag{3.10}$$

The recursive relationship is linear with the following action

$$A_F \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix} \rightarrow \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} \tag{3.11}$$

where b_n is computed with (3.10). The $n \times n$ matrix, A_F, has the form

$$A_F = \left(\begin{array}{c|c} \vec{0} & I_{n-1} \\ \hline 1 & \vec{h} \end{array}\right) \tag{3.12}$$

where 0 is an $(n-1) \times 1$ zero vector, I is the $(n-1) \times (n-1)$ identity matrix and $\vec{h} = (a_1\ a_2 \cdots a_{n-2}\ a_{n-1})$ is the $1 \times (n-1)$ vector of polynomial coefficients. The indices of the non-zero entries of $\vec{h}$ are called *tap points.* Primality implies that 0 is always a tap point since $a_0 = 1$ is a necessary requirement for an irreducible polynomial. A matrix of the form (3.12) whose tap points correspond to a primitive polynomial, which is used to step a shift register forward is called a *Fibonacci* register. The following is true of a Fibonacci register:

1. The matrix A_F in (3.12) is periodic and has period $2^n - 1$

$$A_F^{(2^n-1)} = I_n.$$

2. The sequence of register states

$$S = \{A\vec{r}_0, A^2\vec{r}_0, ..., A^{2^n-1}\vec{r}_0\}$$

is a group under the binary operation $(\vec{r}_i, \vec{r}_j) \rightarrow A^{(i+j)}\vec{r}_0$ for $\vec{r}_i, \vec{r}_j \in S$.

3. S is isomorphic to the multiplicative group of nonzero elements of the finite field $GF(2)[x]/(p(x))$

$$S \cong (GF(2)[x]/(p(x)))^{\times}.$$

4. The sequence of most significant and/or least significant bits of S is a high quality random binary bit sequence.

As a specific example, consider the primitive trinomial

$$p(x) = 1 + x^3 + x^5 \tag{3.13}$$

this defines the recursion

$$r_5 = r_0 + r_3 \tag{3.14}$$

which has the Fibonacci stepping matrix

$$A_F = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.15}$$

stepping the shift register forward has the following action

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_0 + b_3 \end{pmatrix}. \tag{3.16}$$

In terms of logical operations, the 5-bit Fibonacci register can be accomplished by summing the values of the shift register at the tap points, reducing this modulo two, left-shifting the register one bit towards the MSB, and placing the resultant sum in the LSB position. To see how the Fibonacci register acts on a 5-bit integer, consider the integer $28 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$. Then $A_F 28 \rightarrow \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = 25$.

The bottom row of (3.12) is an n-bit binary sequence which can be interpreted as an n-bit integer whose MSB is to the left of the LSB. The bottom row of (3.12) will play an important role in the discussion of the Galois register. Therefore, it is given the following definition

Definition 3.3.4. *The mask, M_F, of a Fibonacci register corresponding to the primitive polynomial $p(x)$ is the bottom row of the matrix (3.12).*

The mask is 1 at tap points and 0 elsewhere.

For the present effort, a 32-bit binary shift register is used which corresponds to the primitive polynomial

$$p(x) = 1 + x^{25} + x^{27} + x^{29} + x^{30} + x^{31} + x^{32}. \tag{3.17}$$

The Fibonacci mask associated with this primitive polynomial, expressed in hexadecimal, is

$$M_F = 0x80000057. \tag{3.18}$$

When using Fortran 90 bit-wise intrinsics, the MSB corresponds to element 31 of the binary number M_F and the LSB corresponds to element 0. Thus using the appropriate Fortran indexing, the tap locations corresponding to the polynomial (3.17) are 31, 6, 4, ,2, 1, 0. The Fibonacci register is logically implemented in Algorithm 3. Given an initial fill of the 32-bit register, each call to Algorithm 3 advances the state of register and returns the MSB of the previous state. Successive calls to Algorithm 3 results in a random binary bit sequence which can be used to bit-wise initialize the canonical rectangle of an ALFG register.

3.3.3 Galois Register

The *Galois* register is derived from the Fibonacci register by taking the transpose of the matrix representation for the Fibonacci register,

$$A_G = \left(\begin{array}{c|c} \vec{0} & 1 \\ \hline I_{n-1} & \vec{h} \end{array}\right) \tag{3.19}$$

where $\vec{0}$ is $1 \times (n-1)$ and $\vec{h}$ is $(n-1) \times 1$. The Galois mask, M_G, is the same as the Fibonacci mask, M_F, but plays an important role in the

Algorithm 3 Fibonacci register implementation. Successive calls generates a random binary bit sequence.

```
1: common integer F
2: integer TAP = ( 31, 6, 4, 2, 1, 0 )
3: bit ← 0
4: j ← 1
5: while (j ≤ length(TAP)) do
6:    bit ← bit + getbit(F, TAP(j))
7: end while
8: bit ← iand(bit, 1)
9: F ← leftshift(F)
10: if (bit) then
11:    F ← bitset(F, 0)
12: end if
13: return bit
```

logical implementation of the shift register on a computer. To understand the action of the transformation on a shift register, consider the 5×5 Galois step, derived from the primitive trinomial (3.13), applied to an arbitrary shift register

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} b_4 \\ b_0 \\ b_1 \\ b_2 + b_4 \\ b_3 \end{pmatrix} = \begin{pmatrix} 0 \\ b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} + b_4 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}. \tag{3.20}$$

Again, taking the MSB of the register to be r_0 and the LSB to be r_{31}, the Galois step is logically implemented in Algorithm 4. The matrix A_G has the same algebraic properties as A_F, however, Algorithm 4 can be implemented more efficiently than Algorithm 3. To see this, let k denote the number of non-zero terms of the primitive polynomial (3.7). The logical operations necessary to advance the register a single step with Algorithm 3 are: perform a k-term summation of tap points, perform a logical *AND* on the resultant, logically *leftshift* the register 1-bit towards MSB, and feed the

resultant bit into the LSB position. This requires $k+3$ operations. The Galois register is efficiently implemented by exploiting the Galois mask. The logical operations necessary to advance the register a single step with Algorithm 4 are: obtain the LSB, logically *rightshift* the register 1-bit towards the LSB, and logically XOR the register with $LSB * M_G$. This requires four operations regardless of the number of tap points. Thus, employing a dense primitive polynomial (which has certain advantages itself) adds no additional compuational expense.

Initializing the canonical rectangle with a Galois register is a novel enhancement which should improve initialization efficiency. In order to compare the initialization times, both registers are used to initialize the canonical rectangle for an ALFG of varying register length. Figure 3.1 shows the initialization time, measured in milliseconds, as a function of ALFG register length. Results are averaged over 100,000 trials and, as expected, initialization times are significatly smaller when using a Galois step.

Algorithm 4 Galois register implementation. Successive calls generates a random binary bit sequence.

1: common integer G
2: integer M_G
3: $bit \leftarrow getbit(G, 0)$
4: $G \leftarrow rightshift(G)$
5: $G \leftarrow G \oplus (bit * M_G)$
6: **return** *bit*

3.4 Statistical Tests

As mentioned previously, empirical tests of RNGs fall into two categories: statistical tests and applications-based tests. Statistical tests are

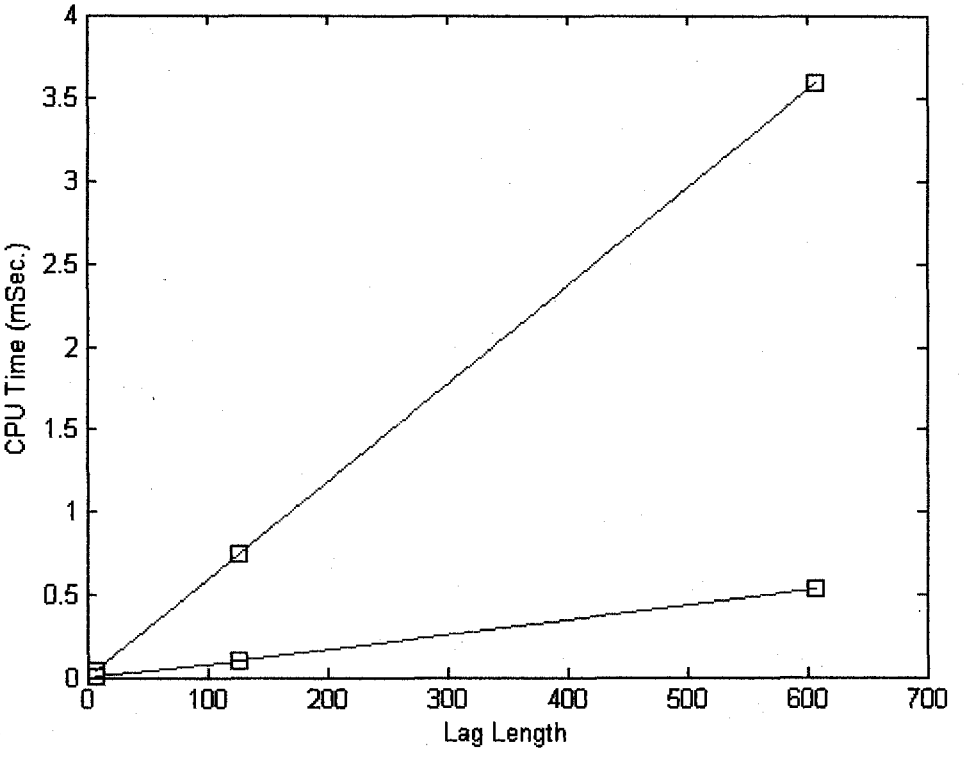

Figure 3.1: Initialization timings for Galois (blue) and Fibonacci (red) shift registers. Timings, measured in milliseconds, are plotted as a function of ALFG lag length.

further subdivided into two categories: *weak* tests and *strong* tests. Weak tests exercise the minimum qualifications necessary for a sequence to be considered random. They are often easy to implement and easy to pass (even for low-quality generators.) While weak tests cannot confirm the quality of a sophisticated RNG, they may quickly rule a generating algorithm out. Computing the mean, standard deviation and higher order moments of a sequence of random numbers are considered weak tests. *Stringent* (or strong) tests are more difficult to implement and more difficult to pass as they are designed to detect subtle defects in RNGs. The stringent tests employed herein are the *Birthday Spacings Test* [38], *Collisions Test*, *Gap Test*, and

Runs Test [31]. These specific tests have been particularly effective in exposing defects in parallel RNGs [60].

Each test is applied to a sequence x_1, x_2,...,x_n,... of real numbers, which purports to be independently and uniformly distributed between zero and one. Some of the tests are designed primarily for integer-valued sequences, instead of a real-valued sequence. In this case, the integer sequence, I_1, I_2,...,I_n,..., defined by $I_n = floor(d\ x_n)$, is a sequence of integers that purports to be independently and uniformly distributed between 0 and $d-1$. The number d is choosen for convenience, e.g. if $d = 2^6$, then I_n might represent 6 specific bits in a 32-bit signed integer.

3.4.1 Birthday Spacings Test

Let the RNG produce m integers I_1,...,I_m in the range $0 \leq I_i \leq d$. The integers can be thought of as m birthdays in a year consisting of d days. The birthday spacings test measures the number of "days" between successive birthdays. Assuming a uniform distribution of birthdays, the set of successive days apart produces a statistic that is asymptotically Poisson with parameter $\lambda = m^3/4d$. Good values for m and d are found to be $m = 2^{10}$ and $d = 2^{24}$. The Poisson distribution function is discrete and therefore discontinuous, thus a KS test cannot be immediately applied. In order to combine a chi-square test with a KS test the following steps are performed:

1. Produce 500 31-bit integer streams of length m.

2. For each of these streams, extract a sequence of m 24-bit integers from the original stream by extracting bit fields 0-23 from each integer in

the original stream. This produces 500 sequences that satisfy the hypotheses of the birthday spacings test.

3. Perform a birthday spacings test on all 500 24-bit sequences. Bin the trials out and compute a single χ^2 value, V_1.

4. Logically shift the bit window one position to the left (towards the MSB) and repeat steps 1-3, this time on bit fields 1-24. This produces the second χ^2 value, V_2.

5. Proceed inductively through bit positions 6-29. In this way, seven χ^2 values, V_1,...,V_7, are obtained.

6. Compute the EDF, $F_7(x)$, from the observed χ^2 values.

7. Perform a KS test by computing the statistics, K_7^+ and K_7^-. If the KS percentile falls within the 95% confidence interval of the appropriate KS distribution, then the test is passed.

3.4.2 Collisions Test

probability	.009	.034	.201	.232	.266	.204	.043	.011
collisions	0-101	102-108	109-119	120-126	127-134	135-145	146-153	$\geq$ 154

Table 3.3: Discrete PDF for the collisions test.

Suppose n balls are tossed at random into m urns. If $m >> n$ then, on average, most balls will land in urns that were previously empty. If a ball falls into an urn which is already occupied, then a *collision* occurs. This test counts the number of observed collisions and compares it to the known distribution which is given in Table 3.3. The observations fall into one of 8 distinct categories, thus a chi-square test is appropriate. As with

the birthday spacings test, the chi-square test must be repeatedly applied in order to apply a KS test. The test is applied to a sequence of $5n$ random numbers by choosing $n = 2^{14}$ and $m = 2^{20}$ and doing the following:

1. Form a sequence of $n = 2^{14}$ 20-bit integers by extracting bit fields 0-3 from 5 consecutive integers of the original sequence and concatenating the binary sequence to form a single integer in the new sequence.

2. Search the new sequence. If an integer is repeated, then a collision is recorded.

3. Repeat the test 100 times on different sections of the stream, bin the trials out, and compute a single χ^2 value, V_1.

4. Logically shift the bit window one position to the left and repeat Steps 1-3 on bit fields 1-4 and compute V_2.

5. Proceed inductively to obtain ten χ^2 values, $V_1, ..., V_{10}$.

6. Compute the EDF, $F_{10}(x)$, for the observed V_i's.

7. Perform a KS test and compute the statistics, K_{10}^+ and K_{10}^-. If both KS percentiles fall within the 95% confidence interval of the associated KS distribution, then the test is passed.

3.4.3 Gap Test

This is a test on a sequence of real numbers $x_1, x_2, \ldots, x_n, \ldots$. This test is used to examine the length of a gap between occurances of a value x_j that falls within a specified range $[\alpha, \beta]$. The true distribution for the outcome is discrete; thus a χ^2 test is applied. As before the gap test is performed on several different sections of the random number stream by doing the following:

1. Apply the test to one-hundred different sections of the random number sequence and obtain one-hundred χ^2 values, $V_1, \ldots, V_{100}$.

2. Compute the EDF, $F_{100}(x)$ from the observed χ^2 values.

3. Perform a KS test by computing the statistics, K_{100}^+, K_{100}^-. If both values fall within the 95% confidence interval of the KS distribution, then the test is passed.

3.4.4 Runs Test

This is also a test on real numbers. The test is very similar to the Gap test and the exact same statistic is computed.

1. Apply the test to one-hundred different sections of the random number sequence and obtain $V_1, \ldots, V_{100}$.

2. Compute the EDF, $F_{100}(x)$ from the observed chi-square values.

3. Perform a KS test by computing the KS statistics, K_{100}^+, K_{100}^-. If both values fall with the 95% confidence interval of the KS distribution, then the test is passed.

3.5 Results

In this section, results of the statistical tests are presented. In order to assess the effectiveness of the parallel initialization we follow Marsaglia [14] and *interleave* parallel streams. Suppose there are *nproc* parallel streams, then streams are interleaved in the following manner: If stream i is given by

$$x_{i,0}, x_{i,1}, ..., x_{i,k}, ... \quad 0 \leq i \leq nproc \tag{3.21}$$

then the new stream is

$$x_{0,0}, x_{1,0}, \ldots, x_{N,0}, x_{0,1}, x_{1,1}, \ldots, x_{nproc,1}, \ldots \tag{3.22}$$

If the parallel streams are independent of each other, then the newly formed stream (3.22) should be random. Any correlation amongst parallel streams manifests itself as intra-stream correlation in (3.22) [60].

In order to assess the quality of the two initialization schemes, both registers are used to initialize parallel streams and the resulting streams are subjected to the suite of statistical tests. The ALFGs studied herein are: $LFG(607, 334, 31)$, $LFG(127, 97, 31)$, $LFG(7, 3, 31)$. If a stream passes a test, then the result is labeled (P), else it is labeled (F).

3.5.1 Single-Stream Results

Each ALFG produces a random number sequence consisting of 10 million numbers. Both the 31-bit integer and real number sequences are collected and analyzed. The results of the statistical tests are summarized in Table 3.4.

$LFG(7, 3, 31)$ fails all of the stringent tests except for the gap test. It does, however, pass the weak tests. Figure 3.2 (left image) shows that as the length of the sequence is increased, the mean value converges to $1/2$, as expected. Figure 3.2 (right image) also shows the defective sequence embedded in the unit square by plotting successive values of the sequence, (x_n, x_{n+1}). This visual test is easy to apply and may quickly identify a suspect RNG. When visualized this way, two-dimensional correlations can manifest themselves as noticeable patterns in the distribution of points. The image is that of a very uniformly distributed set of points in the plane,

however, the stringent tests reveal the presence of higher-dimensional correlations. Figures 3.3 and 3.4 show graphical results of the stringent tests for $LFG(7,3,31)$. All tests except for the gap test produced KS percentiles far outside their respective 95% confidence intervals.

ALFG	Collisions	Runs	Gap	Birthday
LFG(7,3,31)	F	F	P	F
LFG(127,97,31)	P	P	P	F
LFG(607,334,31)	P	P	P	P

Table 3.4: Stringent test results for single-stream ALFGs.

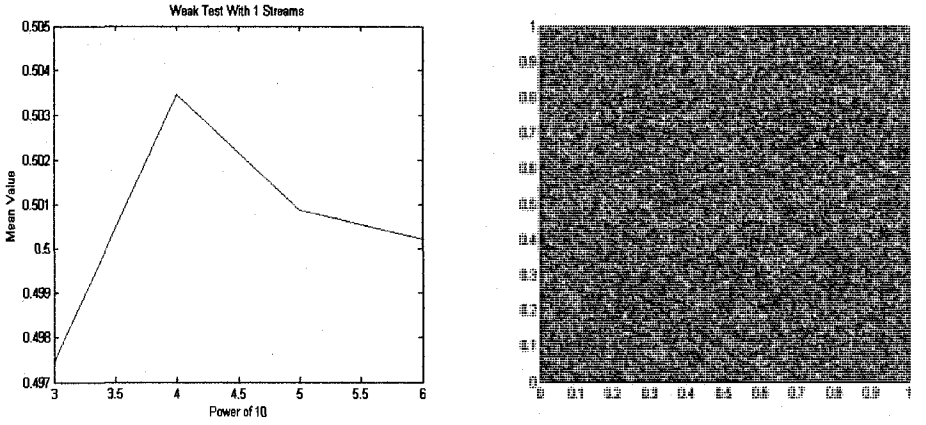

Figure 3.2: On the left, a single stream $LFG(7,3,31)$ converging to the expected mean value of $1/2$ as the length of the sequence increases. On the right, the same stream embedded in the unit square by plotting succesive values, (x_n, x_{n+1}). No visible pattern or structure is apparent.

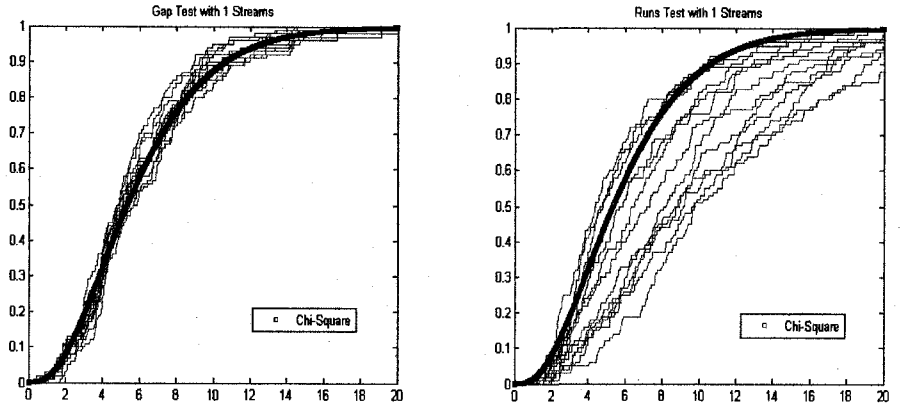

Figure 3.3: On the left, results of the gap test applied to a single stream $LFG(7,3,31)$ are plotted. On the right, results of the run test are plotted. The sequence passes the gap test and fails the run test. The EDFs are depicted in blue and the expected distributions in red.

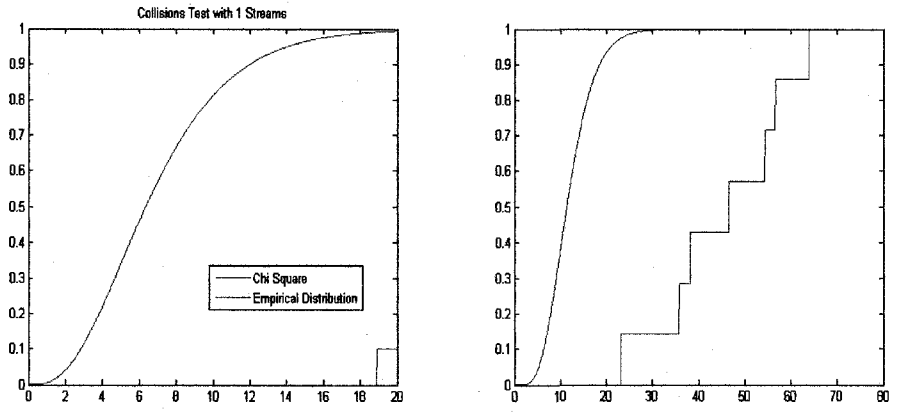

Figure 3.4: On the left, results of the collisions test applied to a single stream $LFG(7,3,31)$ are plotted. On the right, results of the birthday spacings test are plotted. The sequence fails both tests. The EDFs are depicted in red and EDFs in red.

$LFG(127,97,31)$ easily passes the weak tests and passes all stringent tests except for the birthday spacings test. Therefore $LFG(127,97,31)$

is suspect but not necessarily defective. The ALFG is known to fail the birthday spacings test for small values of l ($l \leq 127$) [38]. The figures in Figure 3.5 show graphical test results of the birthday spacings test applied to a single stream $LFG(127, 97, 31)$. On the left, Figure 3.5 shows the EDF compared to the expected distribution for the collected χ^2 values. The KS value computed from these distributions is far outside the 95% confidence interval. On the right, Figure 3.5 shows the results of the birthday spacings test performed on individual bit blocks from which the seven χ^2 values are computed. $LFG(607, 334, 31)$ easily passes all tests.

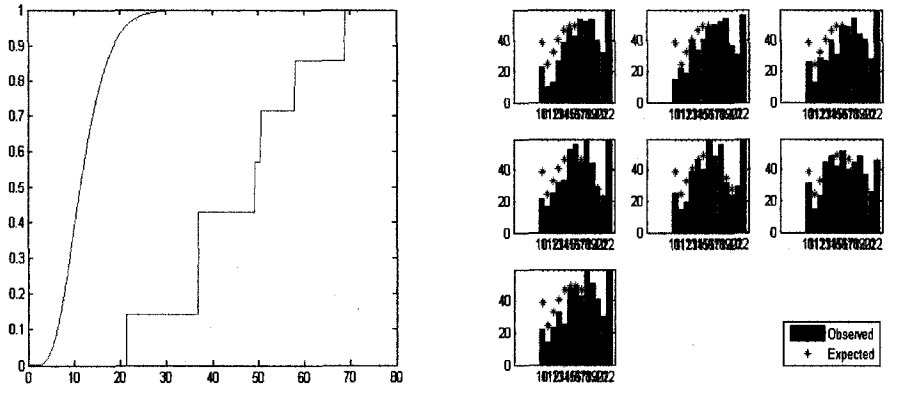

Figure 3.5: Birthday spacings test resulting in failure of a single stream $LFG(127, 97, 31)$. On the left, a KS test is applied to the observed χ^2 values that are obtained from the image on the right which shows seven observed distributions, compared to the expected Poisson distribution, each one corresponding to a different bit block.

3.5.2 Multiple-Stream Results

For multiple stream testing, only the effective sequential generators, $LFG(607, 334, 31)$ and $LFG(127, 97, 31)$, are considered. Each test is performed twice on a sequence consisting of 10^7 random numbers. Once initializing the streams with a Galois generator and once with a Fibonacci

generator. If *nproc* streams are initialized, then each stream produces a sequence of $\frac{10^7}{nproc}$ random numbers. The *nproc* streams are interleaved into a single sequence and tested. Test results are summarized in Tables 3.5 and 3.6. Parallel streams initialized with a Galois register are statistically indistinguishable from those initialized with a Fibonacci register. All tests are passed regardless of the initialization schemem employed.

Streams	Collisions	Runs	Gap	Birthday
5	P	P	P	P
10	P	P	P	P
50	P	P	P	P
100	P	P	P	P

Table 3.5: Stringent test results for $LFG(127, 97, 31)$ initialized with a Galois register. Test results are identical when initialized with a Fibonacci register.

Streams	Collisions	Runs	Gap	Birthday
5	P	P	P	P
10	P	P	P	P
50	P	P	P	P
100	P	P	P	P

Table 3.6: Stringent test results for LFG(607,334,31) initialized with a Galois register. Test results are identical when initialized with a Fibonacci register.

Figures 3.6 and 3.7 show stringent test results for a 50 stream $LFG(127, 97, 31)$. Figures 3.8 and 3.9 show test results for a 50 stream $LFG(607, 334, 31)$. In these tests, 50 streams are initialized and, after an initial run-off (an issue addressed in the next section), produce 50 random number sequences each consisting of 200,000 numbers. The sequences are interleaved and tested; all tests are passed.

Several different streams were tested, each using different values of the global seed S_G. This is done to ensure that a pass wasn't simply a patho-

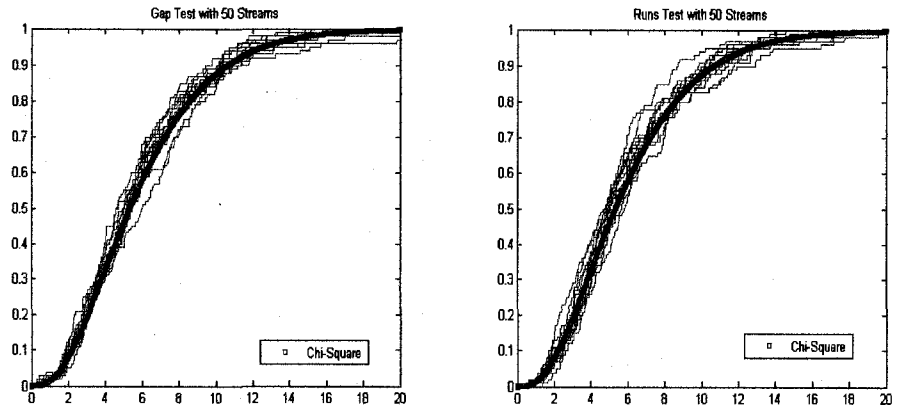

Figure 3.6: On the left, gap test results for an $LFG(127, 97, 31)$ sequences composed of 50 parallel streams. On the right, the results of the runs test on the same sequence. The sequence passes both tests.

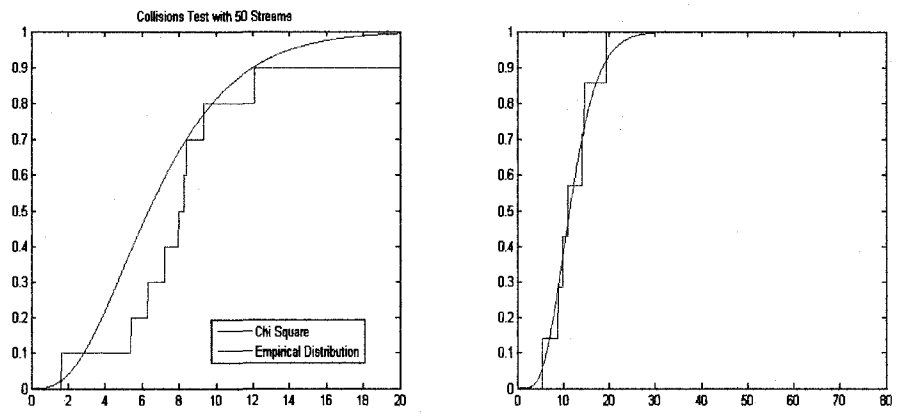

Figure 3.7: On the left, collisions test results for an $LFG(127, 97, 31)$ sequences composed of 50 parallel streams. On the right, the results of the birthday spacings test on the same sequence. The sequence passes both tests.

logical occurance. It was consistently observed that when a test was passed once, it was passed upon every other retest with a different global seed.

Therefore, the empirical evidence strongly suggests that both initialization schemes are effective at initializing high quality parallel streams.

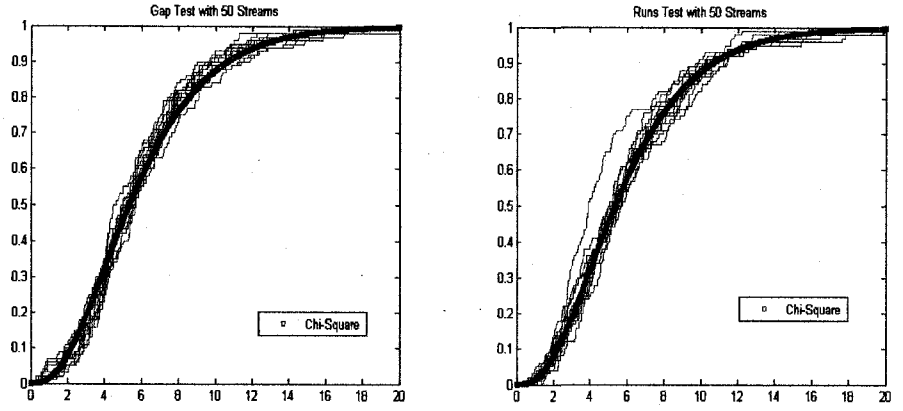

Figure 3.8: On the left, gap test results for an $LFG(607, 334, 31)$ sequences composed of 50 parallel streams. On the right, the results of the runs test on the same sequence. The sequence passes both tests.

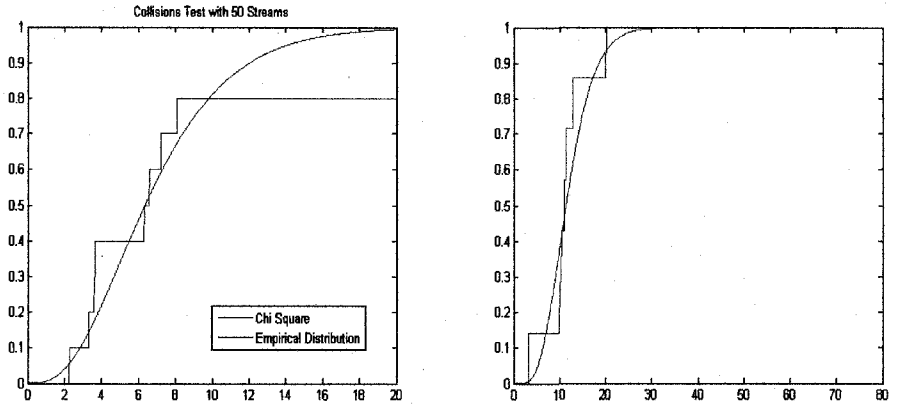

Figure 3.9: On the right, collisions test results for an $LFG(607, 334, 31)$ sequences composed of 50 parallel streams. On the right, the results of the birthday spacings test on the same sequence. The sequence passes both tests.

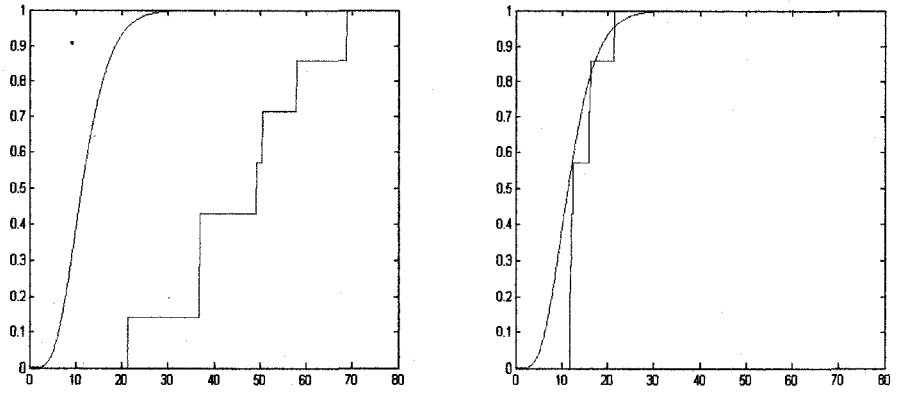

Figure 3.10: On the left, the results of the birthday spacings test on an LFG(127,97,31) stream composed of a single stream and, on the right, results of the same test for a sequence composed of 5 parallel streams. The parallel stream passes the birthday spacing test while the single stream fails.

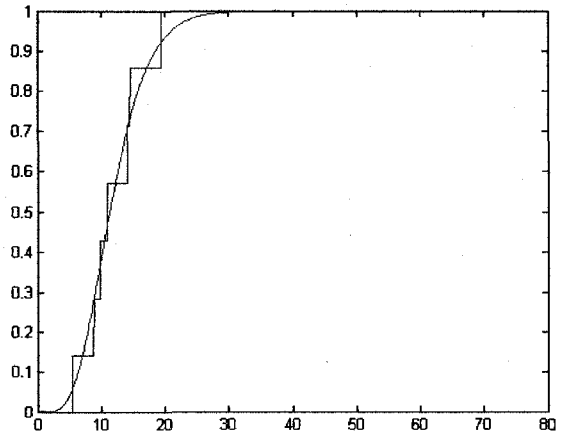

Figure 3.11: Birthday spacings test results on an LFG(127,97,31) stream composed of 50 parallel streams. The parallel stream passes the test.

An interesting observation is the discovery that parallel streams are more random than individual sequences. This phenomenon is observed for all values of l, and k tested, regardless of the initialization scheme employed.

Figures 3.10 and 3.11 depict this phenomenon in the case of the birthday spacings test applied to $LFG(127, 97, 31)$. Figure 3.10 shows the failure of a single stream sequence in the left panel. On the right, an $LFG(127, 97, 31)$ stream composed of 5 parallel streams passes the test. In Figure 3.11 a stream composed of 50 parallel streams is an excellent pass. It has been empirically observed that combination generators perform better than either of the component generators and the performance improves as the number of parallel streams increases. A hueristic argument for this phenomenon is given in [38] and helps explain this observation. Parallel streams, each coming from a separate cycle, can be thought of as a type of combination generator.

3.5.3 Improving Quality

It is also found that the number of transients purged from the generator is important. It is well known that when using the canonical form, initially, the LSB behaves in a less than random way and is the same for all streams on each processor. For all other bit fields, including the next-to-least significant, there is no such defect. Several possibilities exist for remedying this situation: i) the use of a separate, independent generator for only the LSB, ii) shifting off the LSB of the generated random number, so that the integers returned by the generator are in the range $[0, 2^{30} - 1]$ instead of $[0, 2^{31} - 1]$, iii) a huge run-up of the generator in order to purge this defect (empirical studies have shown that about 20 million random numbers must be purged from $LFG(607, 334, 31)$) or iv) simply neglecting this aspect, a viable stratagem when generating random, real numbers where the LSB is not important.

In this work, we choose to implement option (i). This strategy is easy to implement in this situation given that each processor has already initialized a shift register. One solution is to step the shift register with either Algorithm 3 or 4 and replace the least significant bit of x_N with the output bit. The added computational expense is negligible and the enhancement results in immediate gains in accuracy. This particular bit replacement strategy was implemented and the experiment was repeated. All tests were passed at the bit-level, including the birthday spacings test on the low order bits, without any initial run-up.

3.6 Applications-Based Test

The performance and accuracy of the parallel ALFG is further assessed by employing it in an idealized problem in radiative transport. Figure 3.12 depicts, in cross-section, a two-dimensional, Cartesian, prismatic geometry with two internal baffles. Baffles are typically used to collimate particulate transport or to regulate temperature. Only two baffles are modeled in this geometry. The more general situation, where many baffles are likely to be used, follows by induction.

The cross-section of the geometry is square, of unit height and width. The slits made by the infinitely thin baffles are of height 2h, and the baffles are a width 2w apart. The source surface is on the left, and the target surface is on the right. The geometry is vertically and horizontally symmetric - although this fact could be used to reduce the computations, it will not be, as the purpose here is to illustrate the characteristics of the parallel ALFG.

The outputs desired from this geometry are the shape or view factors from the source to the target, F_{st}, and from the source to the second baffle,

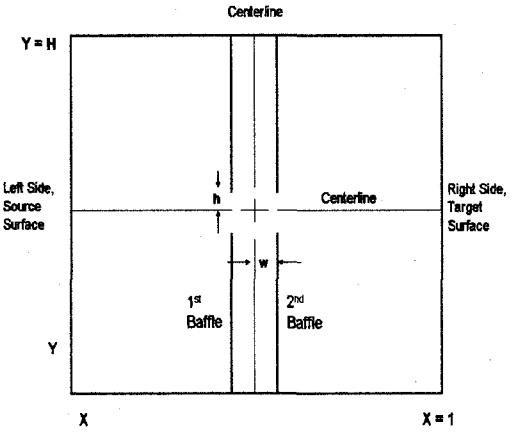

Figure 3.12: Two-dimensional geometry for radiative transfer square with baffles

F_{sb2}. Hottel's crossed-string method is used to compute the view factors, as all surfaces are diffuse and black. This problem is modeled numerically using the well known Monte Carlo ray tracing technique [36]. For small values of w and h, the computation of these view factors by Monte Carlo is very challenging due to the large number of emissions required to achieve accuracy in the very small answers. The emissions from the source surfaces must be distributed directionally and spatially very accurately, as any slight bias or aliasing will manifest in error. Thus, the problem is ideal to illustrate both the accuracy and performance (as many emmissions will be required) of the parallel implementation of the ALFG.

For $w = d = \delta$, applying Hottel's crossed-string method yields:

$$F_{st} = 2\delta(\sqrt{2} - 1) \tag{3.23}$$

and

$$F_{sb2} = \sqrt{\frac{1}{2} + 2\delta^2} - \sqrt{2}(\frac{1}{2} - \delta). \tag{3.24}$$

Applying equations 3.23 and 3.24 with $\delta = 0.005$ yields $F_{st} = 0.0041421$, and $F_{sb2} = 0.0029643$. A typical assumption is that the average value of a large number of samples from most distributions is well modeled by a normal distribution with a specific expected value, μ, and variance, σ^2 [71]. Both F_{st} and F_{sb2} are Bernoulli variables governed by a binomial distribution. Thus for sufficiently large trials, N, the distribution should be approximated by a normal distribution. With this in mind the 95% fractional confidence interval, δ, can be derived from the Central Limit Theorem as [36]:

$$\delta = z\sqrt{(1-F)/FN} \tag{3.25}$$

where F is the current fraction, N is the number of photons emitted and z is the cummulative normal distribution coefficient, which for a 95% confidence interval is about 1.96 [15]. The fractional confidence interval converges to 0 as $N \rightarrow \infty$ thus guaranteeing convergence, albeit at a slow $O(N^{-1/2})$ algebraic rate.

The individual photons do not interact and are thus statistically independent. This leads to a fine-grained parallelism well suited for multiple independent processors [36]. Letting N be the total number of photons emitted, $N_i = \frac{N}{nproc}$ be the number of photons emitted by processor $i = 0, ..., (nproc - 1)$ and let H_i be the total number of hits observed by processor i.

Each task initializes its own independent ALFG stream by following the steps outlined in this chapter. Then Monte Carlo estimates of F_{st} and F_{sb2} are computed by emitting N_i photons per processor. After the emission phase, an all-to-all reduction collective, such as $MPI\_ALLREDUCE$

is called to compute and store $\sum_{i=0}^{num\_tasks-1} H_i$ in each processor's local memory. The approximate fraction is finally computed as

$$F \approx \frac{\sum_{i=0}^{nproc-1} H_i}{(nproc)(N_1)}. \tag{3.26}$$

A floating point, parallel implementaion of $LFG(127, 27, 31)$ has been implemented on the NERSC Bassi system. The implementation generates a global seed, S_G, on $myid = 0$ by calling the *date_and_time*() FORTRAN intrinsic and assigning

$$S_G = (sec+60\times(min+60\times(hr+24\times(day+30\times(mon+12\times yr))))). \tag{3.27}$$

as the global seed which is then broadcast to all processors. Each processor subsequently initializes its binary shift register with the binary number S_G+myid. The register is then stepped $R = 126\times 30$ times with the output bits placed sequentially in each processor's canonical rectangle. The ALFG is not run-up, instead the bit replacement strategy discussed previously is implemented. The parallel ALFG generates 3 random numbers per photon emission and computes F_{st} and F_{sb2} via Monte Carlo.

In order to quantify the efficiency of this implementation, we define the commonly used metric, the *parallel efficiency*, *pe*. Let t_1 be the time to execute an algorithm on a single processor, t_{nproc} the time to execute the same algorithm on *nproc* processors then the parallel efficiency is defined as:

$$pe = \frac{t_1}{t_{nproc} nproc}. \tag{3.28}$$

Parallel timing results for 128×10^7 photon emmissions are shown in Figure 3.13 (left). Figure 3.13 (right) depicts the parallel efficency for up to 128 processors. Broadcast and reduction times are are seen to be negligible

as the parallel performance is observed to be excellent with nearly perfect scaling on 128 CPU's.

The convergence characteristics are seen in Figure 3.14 to be excellent as well. In this figure each processor emits 1×10^7 photons. Results are summed and estimates are computed. In this way, convergence can be viewed as a function of the number of processors. In order to obtain a statistically valid empirical distribution, 1,000 Monte Carlo estimates are computed for a fixed number of processors. Given the expected 95% fractional confidence interval (3.25), the accuracy of the simulation can be assessed by counting the number of approximations that fall outside this envelope. For 1,000 trials, one expects to observe, on average, 50 approximations to fall outside the 95% confidence envelope. As evidenced in Table 3.7, the approximations of F_{st} and F_{sb2} converge to their true values at exactly the rate predicted by the Central Limit Theorem.

Num Tasks	F_{st} Mean Error $\times 10^{-4}$	F_{st} Hits Outside Envelope (Expected = 50)	F_{sb2} Mean Error $\times 10^{-4}$	F_{sb2} Hits Outside Envelope (Expected = 50)
1	0.1739	69	0.2150	53
2	0.1129	48	0.1483	39
4	0.0819	53	0.1052	52
8	0.0590	60	0.0753	63
16	0.0407	52	0.0545	56
32	0.0281	44	0.0384	53
64	0.0203	45	0.0255	48
128	0.0145	47	0.0187	50

Table 3.7: Monte Carlo convergence results; in each trial, active processors emit 1×10^7 photons, 1,000 trials are conducted for a given number of processors. Of the 1,000 trials, we expect, on average, 50 trials to fall outside the 95% confidence interval.

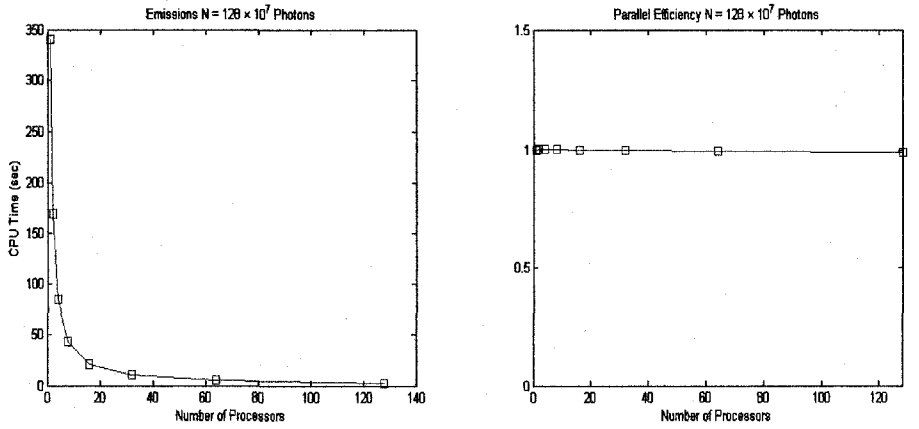

Figure 3.13: CPU timing (left) and parallel efficiency (right) for parallel Monte Carlo photon heat transfer code using 128×10^7 photon emissions.

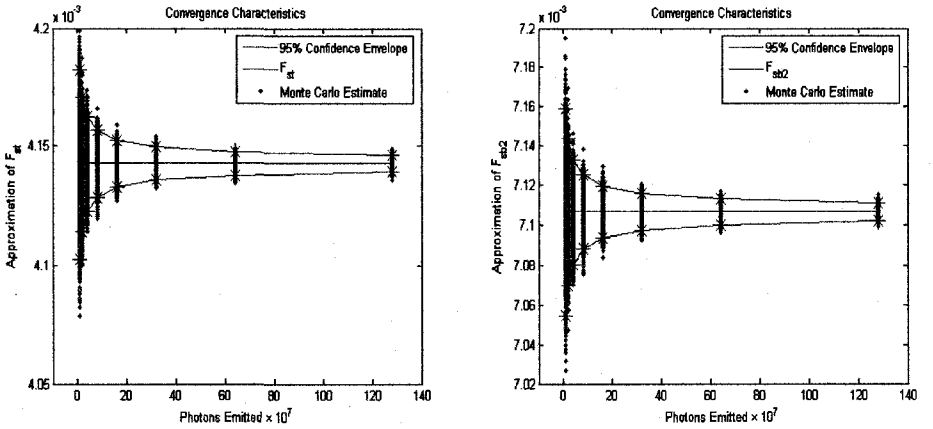

Figure 3.14: Monte Carlo convergence characteristics and theoretically expected 95% confidence intervals for F_{st} (left) and F_{sb2} (right).

3.7 Conclusions

In conclusion, a high-performing, portable, parallel random number generator has been implemented and assessed. It has been demonstrated that multiple, parallel ALFG streams can be effectively initialized using a binary shift register with either a Fibonacci generator or a Galois generator. While both initialization strategies are effective, a Galois generator can be

implemented much more efficiently resulting in initialization times 1/6 of those achieved with a Fibonnaci generator. Furthermore, the statistical characteristics of the parallel RNG streams produced by initializing with a Galois generator are excellent - equally as good as those produced with a Fibonacci initialization.

For sufficiently long register length, the sequential ALFG passes all stringent tests. This result is consistent with past observations. It is also found that interleaved parallel streams are more random than individual streams, a phenomenon explained by the fact that combination generators have been observed to perform better than either of the component generators. Therefore, sequentially sampling across different cycles is a viable strategy for improving a sequential ALFG.

It is observed that the least significant bit is initially not as random as other bits. This is due to the fact that initializing the canonical rectangle results in an initial imbalance in the number of odd numbers in the RNG sequence. An effecive strategy for dealing with this transient defect is to replace the least significant bit by stepping the already initialized binary shift register with either a Fibonacci or Galois step. This effectively and efficiently (especially if using a Galois step) balances the register's even/odd output distribution.

Furthermore, the proposed parallel ALFG has been implemented and executed in a numerically challenging Monte Carlo diffuse radiative transfer simulation. The simulation was run at large-scale on a high-performing, massively parallel architecture. The parallel implementation is observed to scale perfectly and to generate highly accurate results which converge at exactly the statistical rates predicted by the central limit theorem.

Chapter 4

MODELING LARGE ANIMAL POPULATIONS FOR TRACEABILITY SIMULATIONS

According to the USDA, the National Animal Identification System (NAIS) program consists of three components: premises registration, animal identification, and animal tracing. Animal identification provides animal health officials with a starting point for animal tracing. As part of the animal tracing component, producers can choose an Animal Tracking Database (ATD) to record their animal's movements. In the event of an animal disease, epidemiologists will have access to these databases to determine, via traceback, movements and locations of animals involved in the investigation.

As an actual NAIS ATD does not exist, it is necessary to produce an appropriate surrogate or *mock* dataset for the purpose of conducting traceback simulations. This chapter outlines a Monte Carlo process that generates NAIS-compliant mock datasets. In particular, mock datasets are generated with a Monte Carlo discrete events simulator that uses current USDA statistics as input.

This chapter begins with a discussion on probability distribution functions and events probability matrices, continues with the details of the

steps used to produce an output file, and concludes with a summary of the datasets created.

4.1 Theoretical Framework

4.1.1 Probability Distribution Functions

The output of the simulation is based upon probability distribution functions (PDFs) that describe the probability of occurances of discrete events and probability matrices that describe animal transactions which change the state of animal (e.g. birth, death, transfer into and out of premises). A discreet PDF can be defined mathematically in the following way, let $H(x - N)$ be the unit Heaviside function shifted N units to the right

$$H(x-N) = \begin{cases} 0 & \text{if } x \leq N \\ 1 & \text{if } x > N \end{cases} \tag{4.1}$$

and define the interval function $H_{NM}(x) = H(x - N) - H(x - M)$ for $N < M$, which is simply 1 for $N \leq x \leq M$ and 0 elsewhere. A general discrete PDF describing k events is defined in terms of the interval function as

$$F(x) = \sum_{i=1}^{k} f_i H_{N_{i-1}N_i}(x) \tag{4.2}$$

where $N_0 < N_1 < N_2 < ... < N_{k-1} < N_k$ define k discreet bins of width $w_i = N_i - N_{i-1}$. The values of $F(x)$ are constant within each bin. Each bin describes the fraction, f_i, of occurances between the numbers N_{i-1} and N_i. For example, if the PDF describes animal premises sizes, then bin 3 specifies the fraction of premises f_3 that exist of sizes between N_2 and N_3. In general, bin i describes the fraction of observations f_i that occurs between numbers N_{i-1} and N_i. The first bin always begins at $N_0 = 0$. Some properties and constraints that must be observed are

1. Each PDF has a user selectable number of bins, k.

2. The spacing of the N_i is arbitary and need not be uniform.

3. $f_i \geq 0$ for all i.

4. The sum of all f_i must equal 1 or 100%.

4.1.2 Animal Transactions and Animal Events

The mock datasets consists of animal events, wherein an animal may change its state. Examples of animal events include: birth, tagging, death, sold, purchase, etc. For a detailed list of the actual 15 animal, NAIS events modeled see [52]. Herein, animal states are determined probabilistically using *transactions* that may result in one or more events, to allow for animal movement. For example, a transaction might be an animal being sold and consequently purchased - in this example, the animal existing on one premise is *sold* (first event) and then it is *purchased* and moved (possibly) to another premise (second event). Each such event is written to the output file as a row of data. In this example, two rows (events) are written to the output file as a result of a single transaction. Thus the output file created is an events output file.

In the simulation, animal events are mapped to each animal transaction. Event types modeled in this research are from the NAIS specifications cited in [44]. Of the 15 animal, NAIS events modeled only five are of particular interest for the present problem. The *birth* event, the *in* event, the *out* event, the *transition to feedlot* event, and the *died on premises* event. Each animal is "born" on a premises with a *birth* event. Some animals *die on premises* and are removed permanently from the simulation population.

Of particular interest, however, are the *in* and *out* events as well as the *transition to feedlot* event which all result in animal movement. Animal transactions which result in animal movement are important for traceability simulations as these events induce an interconnected "social network" through which a pathogen may travel.

The number of transactions an animal undergoes in a lifetime is determined probabilistically with a transactions PDF which describes the probability that an animal undergoes some number of transactions. Furthermore, to add fidelity, the events probabilities are modeled as varying with the number of transactions. For example, an animal undergoing many transactions is more likely to *die on premises* than an animal undergoing only a few transactions.

Events are modeled using an events probability matrix. The rows of the matrix are the number of animal transactions and the colums correspond to the animal event probabilities. As an example, assume that the total number of events is 3: *purchased*, *slaughtered*, *died*. Assume further that the maximum number of events, excluding birth, is 2. Then Table 4.1 gives the probability of each event for the number of transactions an animal undergoes. The events probability matrix for this scenario would be:

No. of Transactions	purchased event	slaughtered event	died event
1	0.0	0.90	0.10
2	0.20	0.60	0.20

Table 4.1: An events probability matrix

$$\begin{pmatrix} 0.0 & 0.9 & 0.1 \\ 0.2 & 0.6 & 0.2 \end{pmatrix} \tag{4.3}$$

4.1.3 Algorithmic Approach

The simulation generates an animal's event history by probabilistically determining a birth premises, birth date and the number of transactions the animal undergoes in its lifetime. For each transaction, one or more animal events is defined via an events probability matrix. The animal events trace the history of the animal's movements from birth up to either death or the present time.

The simulation requires the following user input parameters for each distinct population type:

1. The total number of animals, na, of a particular type .
2. A PDF, $A(x)$, that characterizes animal age.
3. A PDF, $T(x)$, that characterizes the number of transactions an animal undergoes.
4. A PDF, $P(x)$, that characterizes premises sizes.
5. An events probability matrix M which characterizes animal events in terms of the number of transactions.

Collectively, these inputs completely characterize an animal population. The simulation writes a mock ATD of animal events by implemeting the following steps for each of the NA animals:

1. A birth event occurs and an animal age is selected from $A(x)$.
2. The animal's birth premise is selected from $P(x)$.
3. The birth event is written to the output file.

4. The number of transactions the animal undergoes is selected from $T(x)$.

5. Events are mapped, via M, to each transaction.

6. Each event is processed and written to the output file.

7. If the event is a *death* or *missing*, then processing for that animal ceases.

4.2 Characteristics of Data Used in This Study

Input data to construct the PDF's is obtained from [64]. Because of their differences in ages and premises-movement characteristics, 3 types of animals are simulated: mature cows, preweaned calves, and feedlot cattle.

Age and transaction PDFs, $A(x)$ and $T(x)$ respectively, for mature cows are depicted in Figure 4.1. Premises placement PDF, $P(x)$, for mature cows is shown in Figure 4.2, and the event probability matrix, M, for mature cows can be seen in Figure 4.3. For this population, the fraction of the mature cow population between 365 and 730 days old is 0.22 or 22% of the total mature cow population. The fraction of the population undergoing 3 transactions is 0.32 or 32%. The total fraction of premises with mature cow populations between 0 and 10 animals is 15%. This does not mean, however, that 15% of the mature cow population reside on premises of this size. A simple calculation shows that, in fact, only 1.72% of the mature cow population is resident on premises within this size range.

The PDFs, $A(x)$, $T(x)$, $P(x)$ and events probability matrix M which characterize the preweaned calf population are depicted in Figures 4.4, 4.5 and 4.6 respectively.

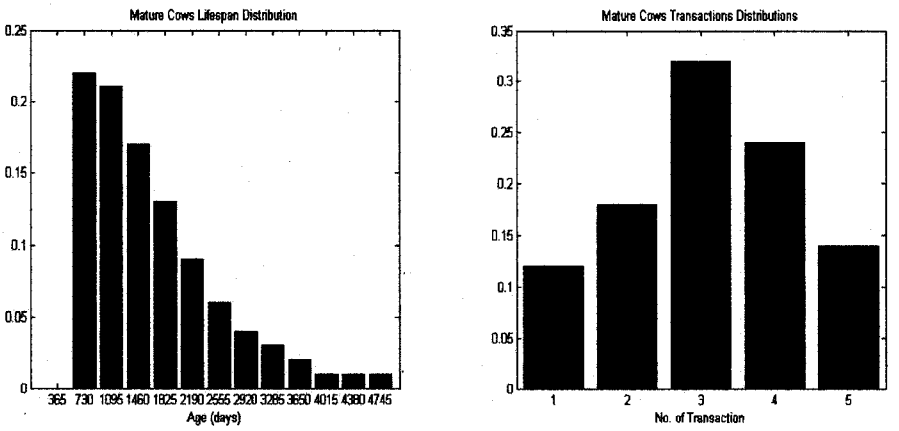

Figure 4.1: Mature cow ages PDF (left) and transactions PDF (right).

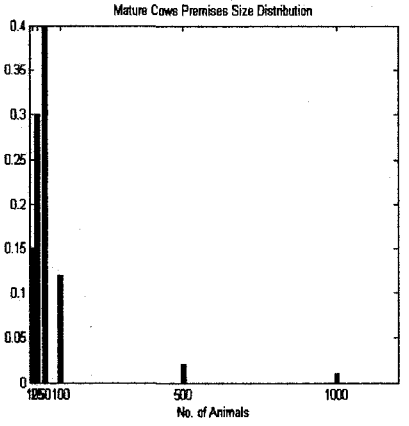

Figure 4.2: Mature cow premise placement PDF.

Transactions	Purchased	Slaughtered	Died	Missing
1	0.900	0.05	0.025	0.025
2	0.800	0.150	0.025	0.025
3	0.700	0.250	0.025	0.025
4	0.600	0.350	0.025	0.025
5	0.500	0.450	0.025	0.025

Figure 4.3: Events probability matrix for mature cattle.

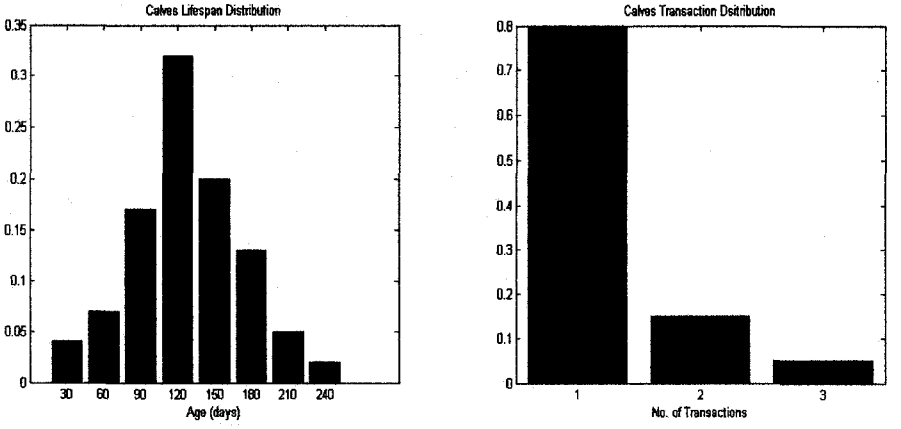

Figure 4.4: Preweaned calf ages PDF (left) and transactions PDF (right).

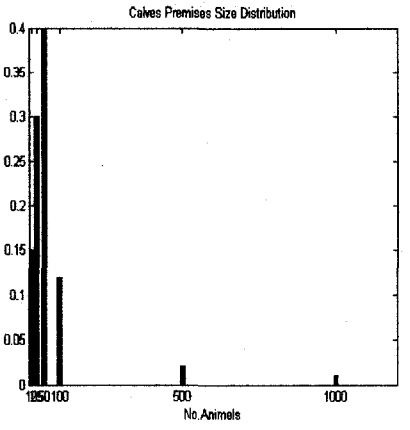

Figure 4.5: Preweaned calf premise placement PDF.

Transactions	Tagging	Purchased	Slaughtered	Died	Missing
1	1.00	0.00	0.00	0.09	0.01
2	0.00	0.90	0.00	0.09	0.01
3	0.00	0.90	0.00	0.09	0.01

Figure 4.6: Events probability matrix for preweaned calves.

PDFs and event probability matrix for feedlot cattle are depicted in Figures 4.7, 4.8 and 4.9. It is noted that all animals, including feedlot ani-

mals, are born and placed onto a birth premises. However, feedlot animals are eventually transitioned to feedlot premises unless they *die* or are *missing*. Mature cows and preweaned calves can be purchased, and move among premises.

4.3 Parallel Implementation

The events simulator is easily parallelized due to the independent nature of the underlying Monte Carlo process. Animal event generation can execute concurrently and independently. The total number of animals to be simulated is evenly distributed among processors. Each processor reads the input file containing PDFs and events matrices. The current implementation is simple with respect to parallel I/O; each processor writes the events it is responsible for to a local output file. A more sophisticated implementation could write to a single, global output file. A complicated parallel I/O model is not necessary for the present effort, as local files can simply be concatenated using the unix "*cat*" command. However, this can be accomplished, for example, with MPI-I/O (a common parallel I/O API). The simulation is parallelized as follows:

1. Each processor reads the input file containing the PDFs and events matrices.

2. Each processor initializes an ALFG stream (see Chapter 3).

3. Processors concurrently execute the same instructions on disjoint sets of animals and writes the output to a local, private file.

4. The local files are concatenated into a single, large file with the unix "*cat*" command.

Parallel execution is independent and, with the exception of the ALFG initialization, requires no communication. It is therefore expected to scale linearly. Table 4.2 contains parallel performance and timing results of the simulation for 2 million animals and verifies the expected performance.

Processors	Time (sec)	Parallel Efficiency
1	79.45	1.0
2	40.12	0.990
4	20.27	0.979
8	10.16	0.977
16	5.05	0.983

Table 4.2: Parallel performance results simulating 2 million animals.

4.4 Verification of Output

The output of the Monte Carlo discrete events simulator is verified by comparing the statistics of a simulated population with the input PDFs. Results from runs simulating a population consisting of 2 million animals generated on 5 processors of the NERSC Bassi system are presented. In this simulation, 800,000 mature cows, 700,000 preweaned calves and 500,000 feedlot cows are simulated. Figure 4.10 shows the age (left) and number of transactions (right) PDFs for the simulated mature cow population dataset. In Figure 4.11, the size of every birth premises is plotted above the premises ID number. All results are in excellent agreement with the input PDFs for mature cows.

Figure 4.12 shows the age (left) and number of transctions (right) PDF for preweaned calves, and Figure 4.13 depicts the same results for feedlot cows. Figure 4.14 shows the population of every feedlot premises plotted above the premises ID. The population is recorded only after all feedlot

animals have been transitioned into the feedlots. Again, all results are in excellent agreement with the input PDFs.

4.5 Datasets Produced

For traceback modeling, several datasets of varying size have been generated. The largest of these mock datasets contains the records for 100 million animals resident on over 1.5 million premises consisting of over 200 million records. All datasets are simulated with the same input PDFs. The datasets are summarized in Table 4.3. The files range in size from 0.50 GBytes for 2 million animals up to 24.0 GBytes for 100 million animals.

No. Animals	No. Premises	No. Events	Filesize (GBytes)
2 million	35,433	4,290,418	0.5
10 million	187,706	21,045,631	2.5
20 million	354,323	42,905,964	5.0
50 million	938,532	105,218,330	12.0
100 million	1,771,616	214,516,041	24.0

Table 4.3: Summary of datasets generated for subsequent traceback modeling.

4.6 Conclusions

In conclusion, a parallel Monte Carlo discrete events simulator capable of generating large, NAIS-compliant mock datasets for the purpose of conducting traceability simulations has been presented. The algorithm has been implemented in parallel on the target architecture of Chapter 2. The output of the simulation has been extensively tested and verified for accuracy. The parallel implementation employs the parallel ALFG in the Monte Carlo phase due to the size of datasets generated. Parallel performance of

the Monte Carlo phase is observed to be excellent, the simulation is highly scaleable and capable of generating massive datasets on the target architecture. For the present effort, datasets consisting of 2, 10, 20, 50, and 100 million have been generated.

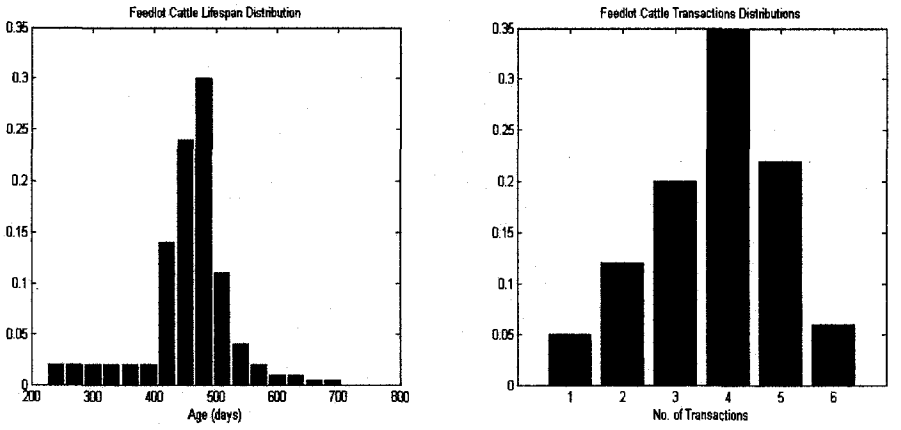

Figure 4.7: Feedlot cattle ages distribution (left) and transactions PDF (right).

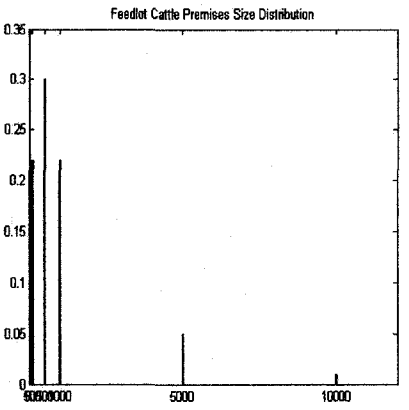

Figure 4.8: Feedlot cattle premise size PDF.

Transactions	Purchased	Slaughtered	Died	Missing
1	0.950	0.040	0.009	0.001
2	0.850	0.140	0.009	0.001
3	0.750	0.240	0.009	0.001
4	0.650	0.340	0.009	0.001
5	0.500	0.440	0.009	0.001
6	0.550	0.540	0.009	0.001

Figure 4.9: Feedlot cattle events probability matrix.

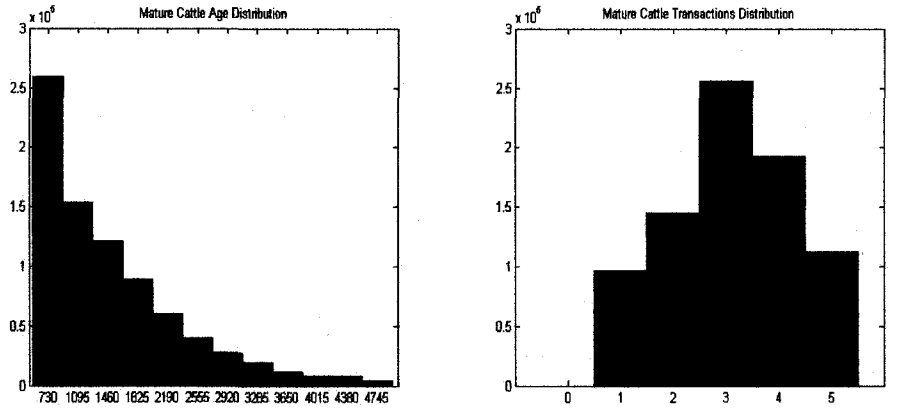

Figure 4.10: Age (left) and transaction number PDFs (right) for a simulated mature cow population. The simulation was run on five processors.

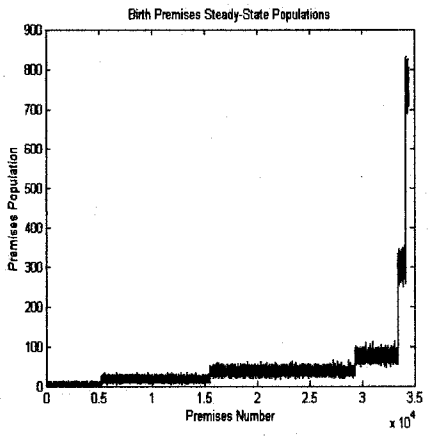

Figure 4.11: Simulated birth premises size distribution.

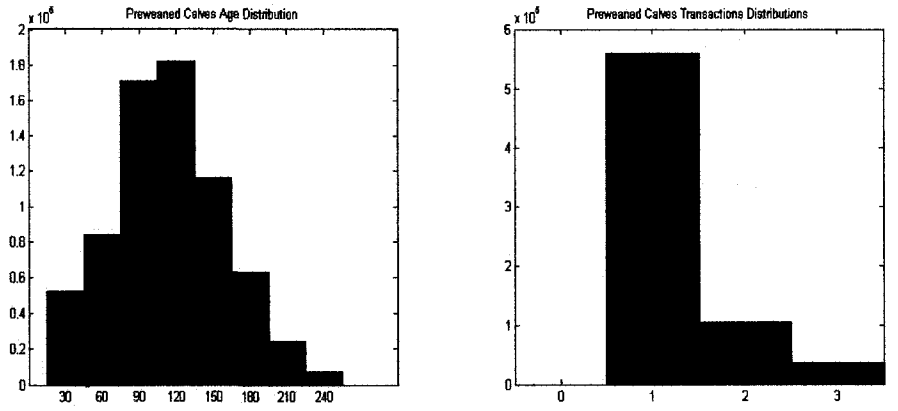

Figure 4.12: Simulated ages (left) and transactions (right) distributions for a population of preweaned calved. Simulation was run on five processors.

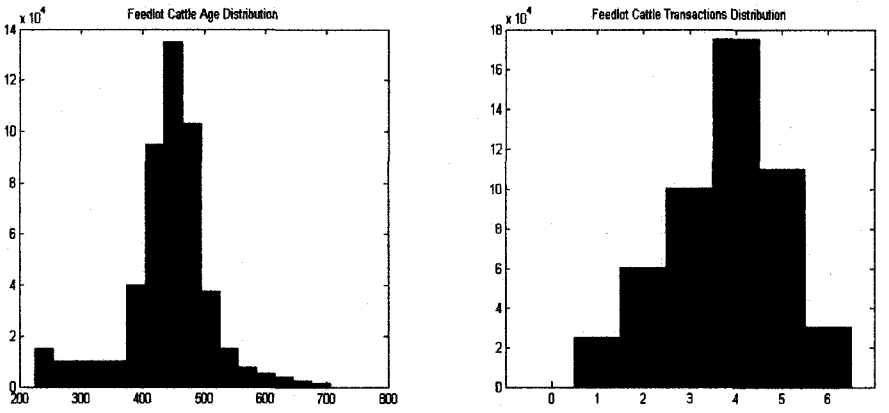

Figure 4.13: Simulated ages (left) and transactions (right) distributions for a population of feedlot cows. Simulation was run on five processors.

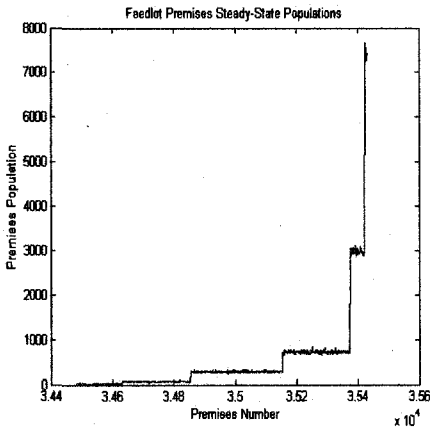

Figure 4.14: Simulated feedlot premises size distribution after all animals have transitioned.

Chapter 5

MEASURED PERFORMANCE AND SCALABILITY OF DATASET TRACEBACK PROCESSING

Traceability is defined as the ability to trace (identify and measure) all the stages that led to a particular point in a process that consists of a chain of interrelated events. Diesease tracing can be thought of as a two step process, *traceback*-from where did the disease come? and *trace forward*-to where has the disease spread? Here, "traceback" will be used generically for both functions, back and forward. In 2008, the current U.S. food supply contains some 108 million head of cattle [65]. Achieving rapid and accurate traceback on a "National" dataset clearly requires a computer. But the questions remain; what computer resources are required? What types of algorithms and programming models should be used? How do system resources and computational performance scale with population size? In order to answer these questions, the techniques of computational science are applied in this chapter.

Due to the size of the "National" dataset produced in Chapter four, an SMP supercomputer consisting of large, shared memory nodes, each containing a small number of processor cores, interconnected over a high

bandwidth switch is a good candidate architecture onto which a high-performance tracing algorithm may be mapped. In Chapter 2, it is argued that a hybrid parallel programming model is the best way to make efficient use of system memory on such a system. One way to implement a hybrid parallel programming model is to use OpenMP for shared memory multi-threading within a node and MPI to pass messages between and amongst nodes. This approach has been applied to many scientific applications [16, 25, 26, 28, 27, 35]. Recent scientific work uncovers the complexity of the many aspects that affect the overall performance of hybrid programs [12, 33, 50, 48]. For some problems, hybrid performance can be inferior to that of a pure MPI implementation, however, substantial performance gains have been observed for certain problems run on SMP clusters [12, 13, 17, 23, 33]. Indeed, this has been demonstrated in the architectural study conducted in Chapter 2.

The goal of this chapter is to devise an efficient and scalable mapping of a disease tracing algorithm and to exercise it on the mock datasets generated in Chapter four. Of particular interest is the parallel performance as the problem size is scaled-up from 2 million animals through intermediate sized datasets to 100 million animals, a "National" dataset. The objectives in doing this are to characterize the computational requirements of the NAIS, determine the feasibility of a 48 hour traceback, provide empirical evidence to support or deny the claim that high-performance can be achieved with a hybrid programming model, and to identify and address computational performance bottlenecks associated with large-scale tracing algorithms.

In the remainder of this chapter, a tracing algorithm is outlined, a hybrid OpenMP/MPI implementation is mapped onto the target architec-

ture, large-scale traceback simulations are conducted and the parallel performance is measured as the problem is scaled-up in size from 2, 10, 20, 50 and 100 million animals. Finally, performance data are presented and analyses follow.

5.1 An Algorithmic Approach

Tracing algorithms typically require nested loops. Such algorithms represent an important class of computationally intensive scientific applications and pertinent algorithms have been well studied (see [21] for a good overview). These algorithms usually impose various data dependencies that result in the need for frequent data exchange among processors when parallelized.

5.1.1 Definitions

We begin by first making some simplifying assumptions and defining relevant quantities. The term *coresident* is used frequently throughout the rest of the chapter and has a precise meaning. In particular, two animals are coresident if they reside on the same premises at the same time. We distinguish between three basic states of animal health; *Infected* - it has been confirmed that this animal has an infectious disease. *Exposed* - an animal is exposed if it has been coresident with either an exposed or infected animal. *Unexposed* - the animal is neither exposed nor infected. Exposures are categorized by *coresidency level*, this quantity characterizes how many degrees removed an exposed animal is from the infected animal. A coresidency level of 1 implies a direct exposure to the infected animal which has coresidency level 0. In general, an animal with coresidency level i is separated from the infected animal by i degrees and "transmits" $i+1$ level

exposures to its unexposed coresidents. The smaller an animal's coresidency level, presumeably, the higher the risk that it may itself be infected.

The movement of animals over time induces a "social network" through which a pathogen may travel via direct or indirect contact on a common premises. The well known *small world* phenomenon (also known as *six-degrees-of-separation*) makes the following highly likely; most, if not all, animals are exposed at some level, and animals that are exposed have been coresident with several different exposed animals at different times in their lives. The recorded *exposure time* is the earliest over all such exposure times and the recorded coresidency level corresponds to the minimum level over all exposures.

5.1.2 A Basic Tracing Algorithm

Tracing is accomplished in parallel at each level and involves recursive scatter-gathers of animal coresidency levels as depicted in Figure 5.1. Suppose $T(C)$ is a retrival function that returns the coresidents of C, then tracing can be solved by recursion,

$$C_{i+1} \leftarrow T(C_i). \tag{5.1}$$

The initial state is a list containing a single infected animal, C_0. After one iteration, $C_1 \leftarrow T(C_0)$; the new list contains N_1 *primary coresidents* removed one-degree from the infected animal. After two iterations, $C_2 \leftarrow T(C_1)$; the list contains N_2 secondary coresidents. The process continues, in principle, until the null set is returned. In practice, however, if we have not yet reached the null set, then we stop after fourteen iterations.

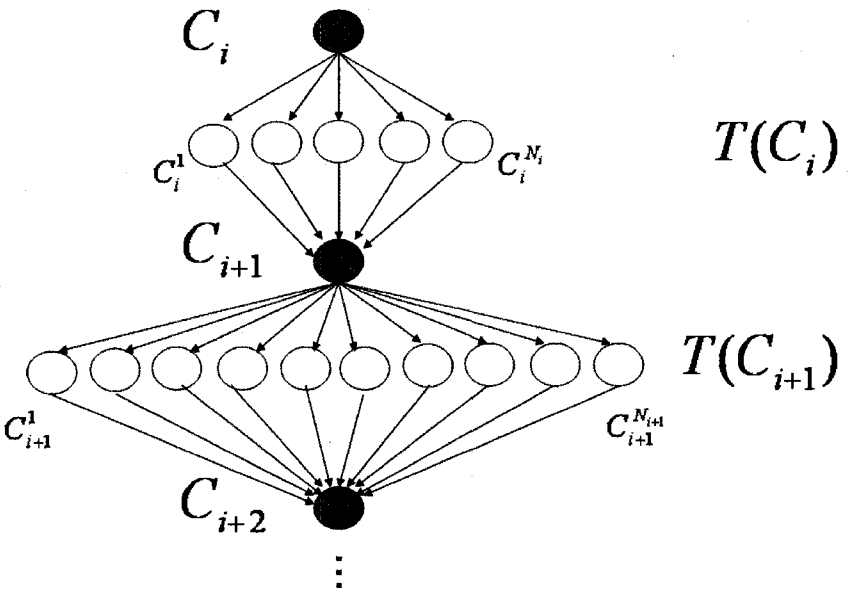

Figure 5.1: A recursive scatter-gather algorithm for disease tracing. C_i is scattered into N_i retrieval tasks. Coresidents are gathered, concatenated and scattered again. Scatter-gather pairs enforce barrier synchronization at the beginning and end of each iteration. In principle, the algorithm proceeds until the null set is returned, in practice we trace through fourteen levels.

5.1.3 Organizing Efficient Data Structures

In this section, the retrieval function is described. Efficient search and retrieval requires precomputing data structures consisting of indexed pointers [1, 63]. In our simulation, events are sequential in time for each animal. It is simple to transform a time sequential dataset into an animal sequential dataset, so without loss of generality, assume the data are sequential with respect to animal identification number, I, where $I = 1, ..., na$ and na is the number of animals tracked in the data file. Each event written to the file is a line of data consisting of a unique animal identification number, I,

Algorithm 5 The full traceability algorithm with all four computational stages: the data is read into memory, data structures are precomputed, the infected animal is traced through L levels, and the results are written to disk. The vast majority of program execution time is spent on the traceback phase which is enclosed within the nested loop.

```
1: program main
2: common A, A*, R, R*, P, X, E_in, E_out
3: read (A, P, E_in, E_out, C_0)
4: R ← sort(P)
5: build R* A*
6: for (i = 0 : (L − 1)) do
7:    N_i ← length(C_i)
8:    for (j = 1 : N_i) do
9:       C_{i+1} ← T(C_i^j)
10:   end for
11: end for
12: write X
13: end program
```

a premises identification number, p, where $p = 1, ..., np$ and np is the total number of premises, and a time of entry into or exit out of p.

After the data file is read into memory, the data are organized into the following data structures; let A^i be an array of animal i.d.'s which is, by construction, already sorted by index in ascending order. If animal 1 undergoes k events in its lifetime, then $A^{1:k} = 1$. Let P^i be an array of premises i.d.'s. Let E^i_{in} and E^i_{out} be arrays containing arrival and departure times respectively of the i^{th} record, where $i = 1, ..., nr$ and $nr > na$ is the total number of data records. Let X be an $na \times 2$ array that contains the coresidency level and exposure time for every animal. Initially, for every animal, the coresidency level is set to "-1" and the exposure time is set to ∞.

Traceback is performed by keying on the premises animals have occupied. Fast retrival of premises i.d. is accomplished by initially sorting the array, P, in ascending order and storing the sorted indices in an array, R^i, where P^{R^i} is monotonically increasing with i. Animal records can then be gathered quickly using the precomputed pointer A_j^*, where $j = 1, ..., na+1$. A_j^* points to the first event address of animal j in array A. Similarly, premises records are available for quick retrieval using the pointer P_k^*, where $k = 1, ..., np+1$. P_k^* points to the first record address of premises k.

Algorithm 5 details the the four distinct computational phases; input, precompute data structures, trace, and output. Algorithm 6 details the retrieval function and its use of the data structures for efficient search and retrieval. Important points to note about this implementation:

1. The algorithm consists of four nested loops.

2. Exhaustive searching is avoided entirely in exchange for a single sort; this reaps large dividends by reducing search and retrival complexity from quadratic to linear complexity.

3. The single sort can be accomplished in $O(nr * log(nr))$ time. Here, we use a heap sort due to its predictable performance.

4. The time it takes to sort the data and precompute the pointers is negligible compared to the time it takes to sequentially perform traceback.

5. The array, R^i, is a permutation of the integers $[1 : nr]$ governed by the input PDFs in Chapter four. For all practical purposes it behaves as a random sequence of integers.

6. While "virtual" premises and event addresses, A^* and P^*, exhibit spatial locality and can be accessed with unit stride, the actual addresses they point to, in general, are distributed throughout memory.

Algorithm 6 The retrieval function $T(n)$ efficiently retrieves coresidents by accessing precomputed data structures.

```
1: function T(n)
2: common A, A*, R, R*, P, X, E_in, E_out
3: count ← 0
4: for (i = A*_n : (A*_(n+1) − 1)) do
5:    for (j = R*_(P^i) : (R*_(P^i+1) − 1)) do
6:       read E_in^(R^j), E_out^(R^j)
7:       if (coresident) then
8:          count ← count + 1
9:          update X^(A^(R^j))
10:      end if
11:   end for
12: end for
13: return coresidents
```

5.2 Mapping the Algorithm

As noted by Foster [20], a "good" parallel algorithm has four fundamental requirements: *concurrency*, *scalability*, *locality*, and *modularity*. A good parallel program strikes a balance between the often conflicting goals of maximizing concurrency, while simultaneously minimizing non-local memory access.

There are two basic parallel computing paradigms: *shared memory* and *distributed memory*. In the shared memory model, processors share a global address space. A shared memory program is a collection of threads of

control which can be created dynamically mid-execution. Threads communicate implicitly by writing and reading shared variables and coordinate by synchronizing on shared variables. In the distributed memory model, the total physical memory is distributed over a collection of processors. Each processor executes the same algorithm under a separate address spaces and thus global variables declared in an distributed memory program are private to each processor. Processors must explicitly communicate with each other through message passing in order to access remote data.

SMP clusters can be thought of as an hierarchical two-level parallel architecture since they combine features of shared and distributed memory machines. On the target architecture, it is possible to mix OpenMP with MPI, and a convincing case is made to do so in Chapter 2. In the following, we first develop a pure MPI parallelization of the tracing algorithm and subsequently extend it to a hybrid OpenMP/MPI model. Our program design approach follows the parallel software engineering methodolgy of Foster [20].

5.2.1 An MPI Approach

MPI is the de facto message-passing standard [42] widely used for high-performance parallel applications and has been implemented on numerous computer systems. In this section, a pure MPI implementation of Algorithm 5 is designed.

Partition Due to the recursive dependence, $C_{i+1} \leftarrow T(C_i)$, it is not possible to partition Figure 5.1 horizontally across levels i. The natural partition is depicted in Figure 5.2 where $N_i \leq na$ retrieval tasks are created at each

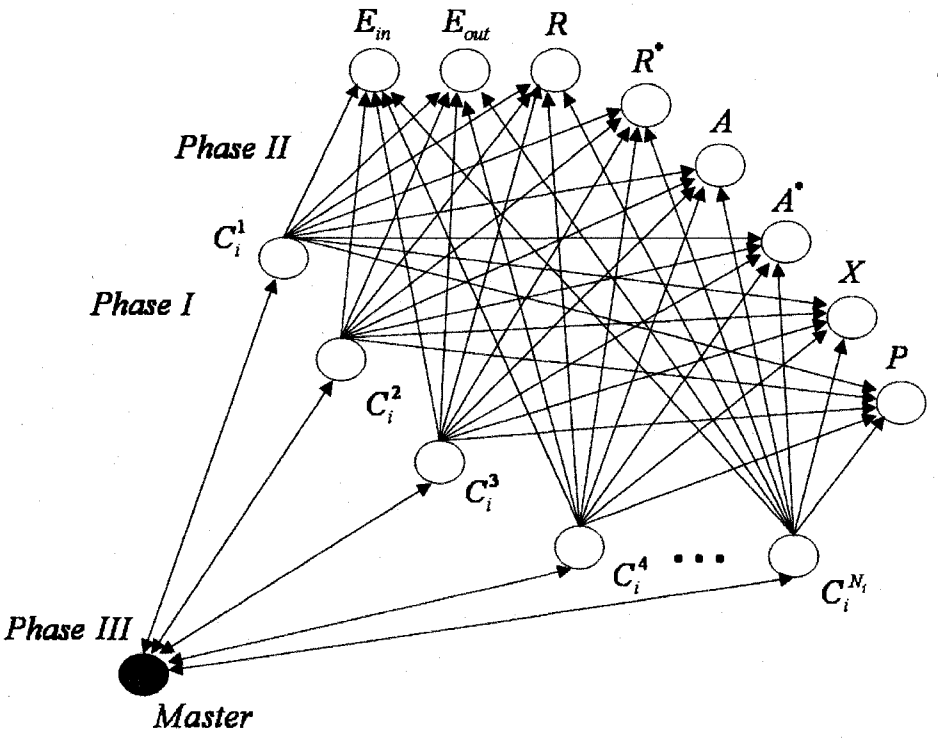

Figure 5.2: Communication channels for a single iteration, $C_{i+1} \leftarrow T(C_i)$. Three distinct communication phases are defined. Phase I scatters C_i into N_i retrieval tasks. In Phase II, retrieval tasks retrieve coresidents concurrently by asynchronously requesting data from data tasks. In phase III, retrieval tasks merge their lists to the master which synchronizes X and prepares to scatter the new list at the start of the next iteration. Scatter-gathers are barrier synchronization events.

iteration, a master task coordinates and schedules their execution, and separate data tasks service retrieval requests.

Partitioning in this manner has several attractive features. It creates a very large number of fine-grained retrieval tasks, several orders of magnitude larger than the number of available processors, all of which can execute concurrently. This partition exposes the maximum opportunity for concurrent execution and allows for the most flexibility in design. Retrieval tasks are of comparable size; each is responsible for applying T to a single suspect

coresident. Retrieval tasks scale ideally; the number of retrieval tasks increase with population size, but the amount of computation each performs remains constant. This partition is not optimal, however, as data tasks do not scale well with problem size. Data tasks grow in size and not in number as population size scales-up.

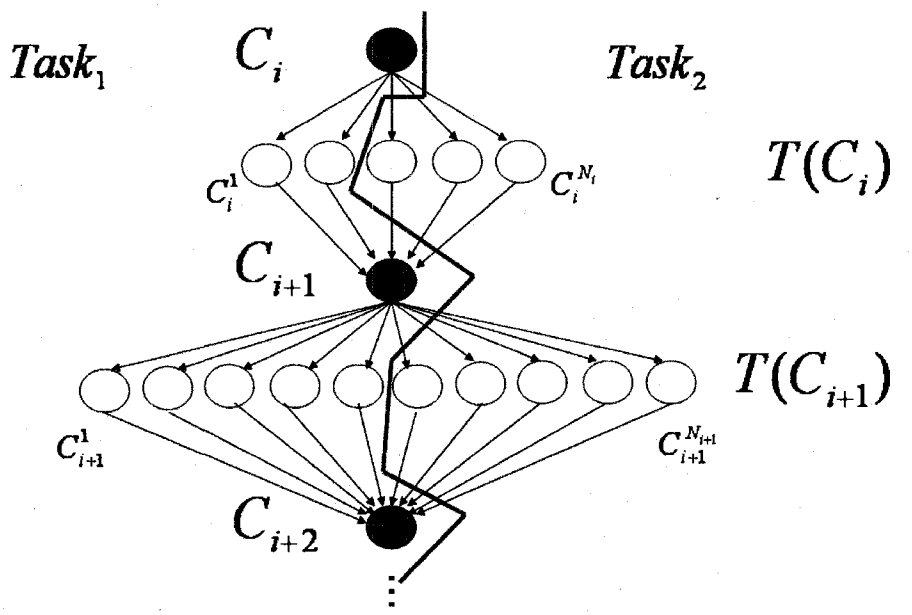

Figure 5.3: Tasks are agglomerated and mapped onto two MPI tasks. $Task_0$ owns the master task and is thus the root for collective communication. Both tasks also function as workers. Each MPI tasks also contains a replicated copy of the data structures required for efficient search and retrieval. Communication paths that intersect the red dividing line require MPI collective communication for data synchronization.

Communication Figure 5.2 provides a detailed depiction of the required commuincation channels and dependencies. Three distinct communication phases are identified. Phase I scatters C_i into N_i retrieval tasks. In Phase

II, tasks concurrently retrieve coresidents via asynchronous data requests serviced by data tasks. In phase III, retrieval tasks merge their lists to the master which synchronizes X, and prepares to scatter the new list at the start of the next iteration. Scatter-gathers are barrier synchronization events.

Communication among retrieval tasks and data tasks is asynchronous, unstructured and voluminous. Retrieval tasks can execute concurrently, however speedups will not scale if data tasks must sequentially process requests from retrieval tasks. This is likely the case in a message passing model which requires two-sided communication. In this case, data tasks must continuously probe for incoming requests and are only able to process a single request at a time. In a shared memory model, this problem is alleviated by one-sided communication however, as seen in Chapter 2, bandwidth contention may degrade performance. Too many retrievals and not enough data tasks to effectively service them is the primary concern for this communication model.

Agglomeration Asynchronous communication is reduced by agglomerating retrieval tasks and a replicated copy of the data tasks into a single MPI task. The memory requirements of each MPI task scales as $O(na)$. The head MPI task ($myid = 0$) plays the role of the master task and scatters C_i into chunks of size $N_i/nproc$ to all MPI tasks (itself included). The retrieval tasks allocated to an individual MPI task are processed sequentially. However, the collection of MPI tasks can process their retrieval tasks concurrently and independently of one-another. When all retrievals have completed, the master MPI task gathers the resultant coresidents, con-

catenates the coresidents into C_{i+1}, synchronizes the state X, and finally scatters C_{i+1} signaling the start of the next iteration.

Data replication is a common technique intended to improve locality with minimal software engineering cost [19, 20]. The principle disadvantage is, of course, that the memory requirements do not scale. However agglomerating this way has some attractive features. In particular, it maximizes concurrency and minimizes communication by increasing locality. Also, by replicating the data, the number of retrievals that can be processed concurrently is increased.

Mapping Each MPI task is mapped onto one of *nproc* physical processors. Conceptually, this mapping is equivalent to partitioning the domain in the vertical direction as shown in Figure 5.3, and assigning each piece to a processor. Arrows crossing the solid, red partition-line requires an MPI collective communication. In this example, $Task_0$ contains the master task and therefore this task is the root for scatter/gather operations. The necessary modifications to Algorithm 5 are made in Algorithm 7. Note the placement of the MPI initialize and finalize directives in the algorithm. A single processor reads the data and precomputes the data structures, and only after these phases are the MPI tasks forked. Once the tracing phase is complete, the MPI tasks are dissolved and only the master task writes the output to disk.

Analysis This mapping is clearly problematic. MPI tasks grow in size far too fast. The memory requirments of this implementation scale as $O(nproc * na)$. This mapping is neither scalable with problem size nor processor count. The total aggregate memory available on the target architecture is about 2

TBytes, however, it is not globally addressable. The sequential algorithm (Algorithm 5) is bound by the amount of memory available on a single node, 32 GBytes. The MPI mapping is bound by 1/8 of the available memory on a node, 4 GBytes. This effectively reduces the range of problems accessible to the MPI mapping to below that of the sequential algorithm. Despite the fact that the mapping maximizes locality and concurrency, it is an inefficient use of memory resources.

Algorithm 7 A pure MPI implementation of Algorithm 5.

1: *program* mpi main
2: *common* $A, A^*, R, R^*, P, X, E_{in}, E_{out}$
3: *read* $(A, P, E_{in}, E_{out}, C_0)$
4: $R \leftarrow sort(P)$
5: *build* R^* A^*
6: $MPI\_Init(nproc, myid)$
7: **for** $(i = 0 : (L - 1))$ **do**
8: $N_i \leftarrow length(C_i)$
9: $MPI\_Scatter(C_i)$
10: **for** $(j = 1 : (N_i/nproc))$ **do**
11: $C_{i+1} \leftarrow T(C_i^j)$
12: **end for**
13: $MPI\_Gather(C_{i+1})$
14: **end for**
15: $MPI\_Finalize()$
16: *write* X
17: *end program*

5.2.2 A Hybrid Approach

In the pure MPI implementation, concurrency and locality have been maximized at the expense of scalability. This inefficient use of memory can be ameliorated by extending the pure MPI implementation to a hybrid

Algorithm 8 A hybrid OpenMP/MPI implementation of Algorithm 5.

1: *program* ompi main
2: *common* $A, A^*, R, R^*, P, X, E_{in}, E_{out}$
3: *read* $(A, P, E_{in}, E_{out}, C_0)$
4: $R \leftarrow sort(P)$
5: *build* R^* A^*
6: $MPI\_Init(nnode, myid)$
7: **for** $(i = 0 : (L - 1))$ **do**
8: $N_i \leftarrow length(C_i)$
9: $MPI\_Scatter(C_i)$
10: !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(j)
11: !$OMP DO SCHEDULE(STATIC)
12: **for** $(j = 1 : (N_i/nnode))$ **do**
13: $C_{i+1} \leftarrow T(C_i^j)$
14: **end for**
15: !$OMP END DO NOWAIT
16: !$OMP END PARALLEL
17: $MPI\_Gather(C_{i+1})$
18: **end for**
19: $MPI\_Finalize()$
20: *write* X
21: *end program*

OpenMP/MPI implementation. In particular, a single MPI task, as previously defined, is mapped to a single SMP node. This reduces the total number of MPI tasks by a factor of eight. Within a node, each MPI task becomes a master OpenMP thread and is responsible for forking a team of retrieval threads which concurrently and asynchronously retrieve coresidents. Node 0 is the master node and the root for MPI scatter/gather operations. In the hybrid model, coresidency levels are scattered and gathered to and from SMP nodes via MPI collectives, while retrievals are processed concurrently within a node via shared memory multi-threading. When all retrieval

tasks complete, control is returned to the master thread which prepares for node synchronization. The necessary extension is found in Algorithm 8, it only requires a single OpenMP do-loop directive judiciously placed.

This approach provides a more efficient use of memory. The memory is again bound to the size of an SMP node, as it is for the sequential algorithm. However, concurrency is increased and the memory requirements scale as $O(na * nnodes)$. Threads can process retrievals concurrently, however this doesn't necessarily imply perfect parallel scaling within a node. As demonstrated in Chapter 2, data access patterns with very large or very small strides can result in shared memory contention and result in poor parallel performance. As population size increases, so too does the number of records and premises. With this we should expect the average access stride to increase as well.

5.3 Numerical Results

In this section dataset processing performance results are presented. The datasets are of the following size: 2 million animals (0.5 GBytes), 10 million (2.5 GBytes), 20 million (5 Gbytes), 50 million (12 GBytes) and 100 million (24 GBytes). For each dataset the hybrid model presented in Algorithm 8 is executed on a variable number of nodes. Each node employs all 8 available threads, unless otherwise stated. The algorithm is implemented in Fortran 90 and is compiled on the target architecture with an IBM *mpxlf_r* compiler. The compiler optimization level is -O4, and the -qsmp=omp flag is set enabling OpenMP. On the target architecture the compiler produces 20 GByte large-page 64-bit executeables. The objective of the code is the same in each numerical experiment; trace a single infected animal, chosen

at random, through 14 coresidency levels, map the coresidency level of each exposed animal, and record the earliest exposure time.

The total execution times reported are the sum of the execution times for the scatter-gather (MPI) phase with the execution time for the retrieval phase (OpenMP). The processor times required to precompute the relevant data structures (pointers) are not included in the timings. The expected scaling versus the size of the data for the heap sort operation [32] is observed and execution times for preprocessing range from 3.9 seconds for 2 million animals up to 13 minutes for 100 million animals. The sequential sorting performance can be improved upon by either parallelizing the heap sort or simply employing an optimized intrinsic such as Fortran's quick sort routine (QSort). Therefore, attention hereinafter is on parallel execution time. Timings presented measure events occuring in between the initialize and finalize MPI directives in Algorithm 8.

5.3.1 Observed Performance

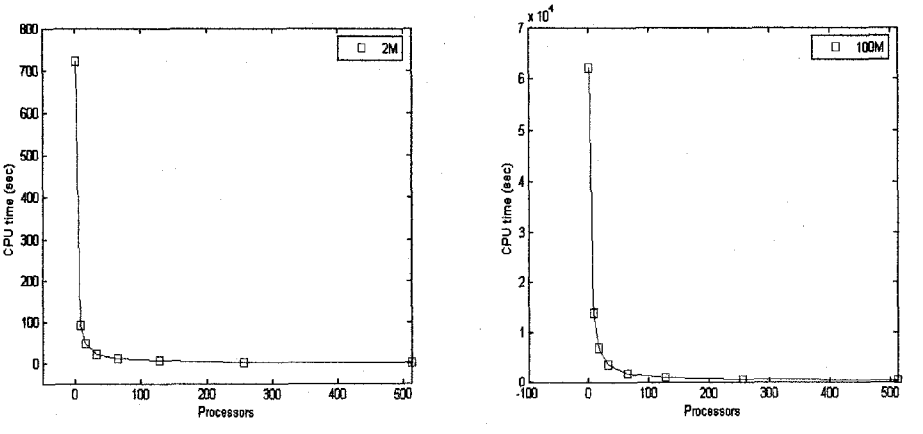

Figure 5.4: On the left, parallel timing results for 2 million animals, and on the right, parallel timings for 100 million animals. Single processor time is 722 sec and 62,039 sec respectively.

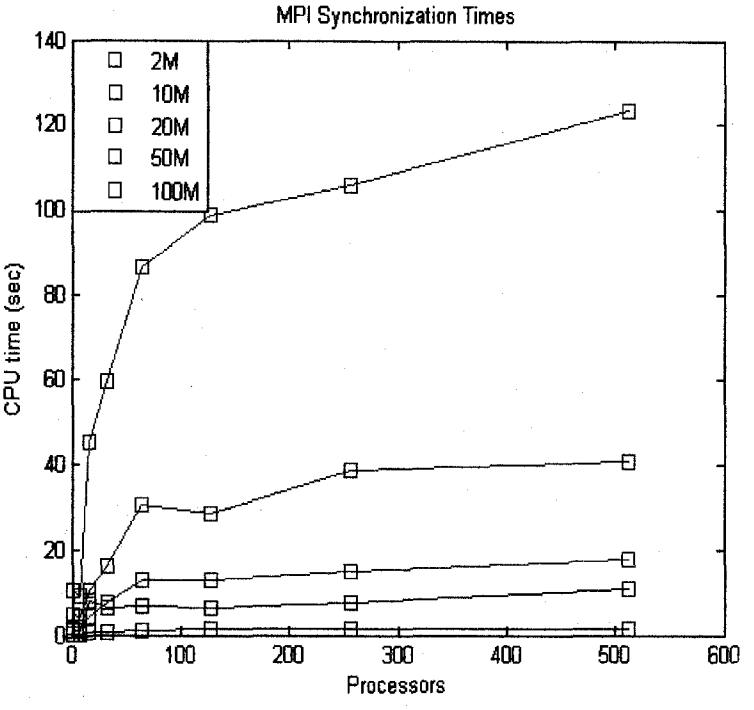

Figure 5.5: Time spent performing global MPI commuincation as a function of processor count. Execution time scales as $O(log(nnodes))$.

Timings Figure 5.4 shows timing results for both 2 million and 100 million animals. Based on raw timing alone, parallel results are excellent. On a single processor it takes 722.11 seconds, or 12 minutes, to trace an infected animal through a population of 2 million animals. Utilizing all 512 available processors, the time is reduced to 3.31 seconds. For 100 million animals the same exercise takes over 17 hours on a single processor. Utilizing all 512 available processors the execution time is reduced to 361.5 seconds, or just slightly over 6 minutes. Intermediate datasets scale linearly with population size, parallel timings follow and timing plots are omitted.

MPI Commuincation Time The execution times presented in Figure 5.4 are obtained by summing the execution time of the retrieval phases,

which decrease with processor count, with the execution times of the scatter-gather phases, which increase with processor count. Barrier synchronization is achieved with global MPI commuincation routines. The communication overhead associated with these opertations is shown in Figure 5.5. The overhead is observed to scale as $O(log(nnodes))$, which is the expected behavior.

In Chapter 2, it is shown that large messages achieve peak bandwidth; in this implementation, messages are na double-precision words (8-bytes) in size. Due to the large message size, MPI communication overhead is dominated by stream rate and not latency. The implementation exploits the high bandwidth HPS switch by hiding the latency associated with collective communication with large messages. MPI communication overhead is observed to be very small in comparison to computation times. Even for the largest dataset, 100 million animals, the total communication overhead on 512 processors is only slightly larger than 2 minutes. Small in comparison to 17 hours on a single processor, however this is significant in terms of the total execution time of about six minutes on all 512 processors.

The parallel tracing algorithm requires global and synchronous MPI communication. In Chapter 2 it is demonstrated that global MPI communication is more expensive than local communication. The hybrid model pays large dividends in this case by significantly reducing the number of MPI tasks; it is much easier to synchronize 64 tasks than it is to synchronize 512 tasks.

Parallel Efficiency and Speedup Raw timings are by no means a complete characterization of performance. Figure 5.6 shows the speedup (top)

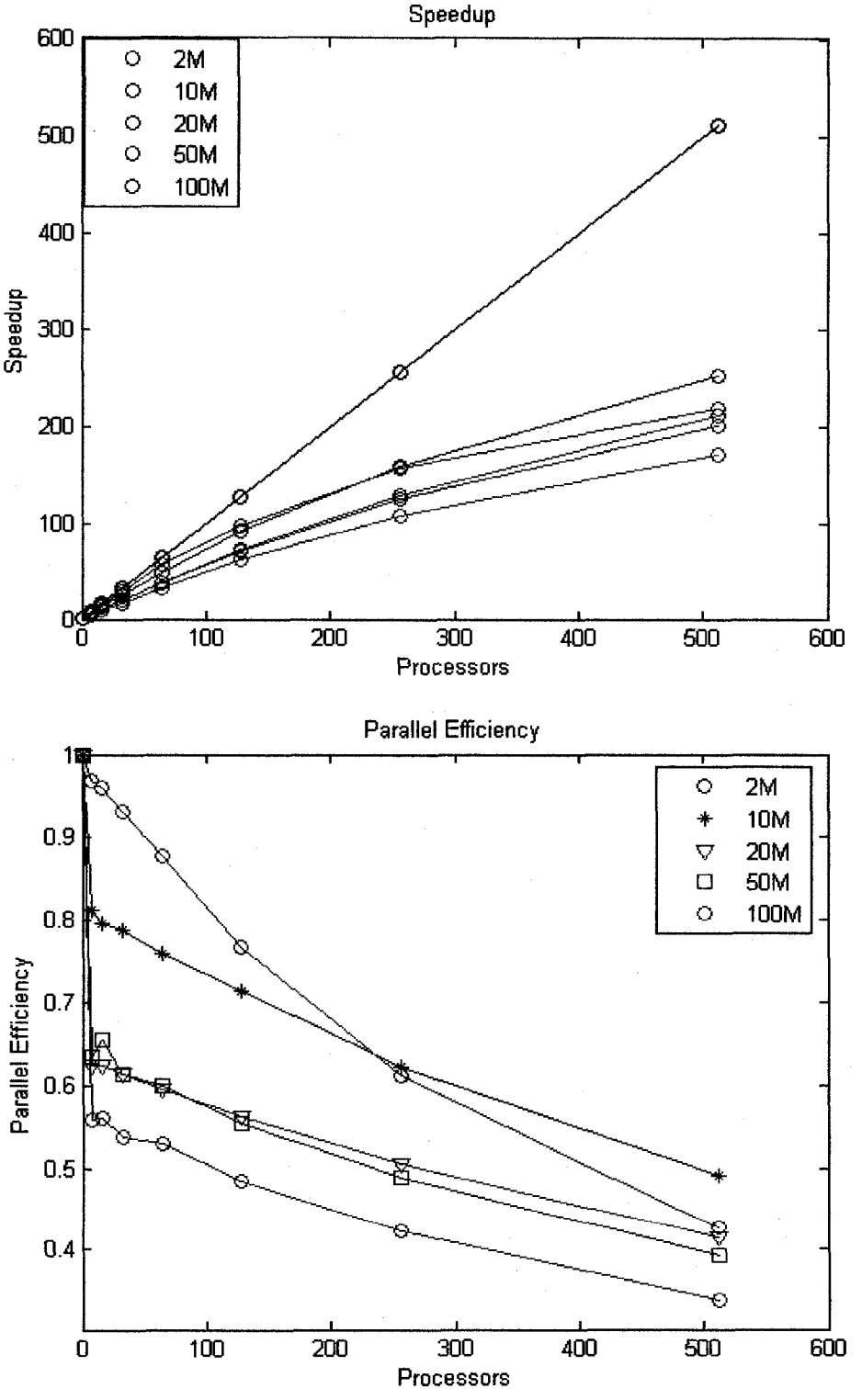

Figure 5.6: Speedup (top) and parallel efficiency (bottom) for the tracing phase of the hybrid implementation.

Nodes	Procs	OpenMP (sec)	MPI (sec)	Total	Parallel Efficiency	Speedup
1	1	62,028.438	10.475	62,038.913	1.0	1.0
1	8	13,865.177	9.354	13,874.531	0.559	4.471
2	16	6,870.025	45.083	6,915.108	0.561	8.972
4	32	3,534.795	59.582	3,594.377	0.539	17.260
8	64	1,738.886	86.679	1,825.565	0.531	33.983
16	128	899.335	98.850	998.185	0.486	62.151
32	256	467.733	105.850	573.583	0.423	108.160
64	512	238.296	123.220	361.516	0.335	171.607

Table 5.1: Parallel performance results for 100 million animals.

and parallel efficiency (bottom) for all sized problems. The data reveal insight not apparent in raw timings. Table 5.2 shows performance data for 2 million animals and Table 5.1 shows performance data for 100 million animals (similar tables for all datasets are found in Appendix B). The far left column of the tables contains the number of nodes and hence the number of MPI tasks. The first two entries in this column are single node results, the first, a single processor and the second, all eight processors within a node. Since no MPI communication occurs with a single node, performance degradation that occurs between the first two entries of the tables is attributed exclusively to shared memory.

The parallel efficiency is particular telling; the hybrid algorithm, when executed on 2 million animals and utilizing all 512 processors, is observed to achieve a 42% parallel efficiency and a speedup of 218. Performance degradation is due almost exclusively to MPI overhead. Shared memory performance is seen to be excellent, indeed nearly perfect at 97% parallel efficiency and a 7.75 speedup. MPI overhead degrades efficiency at a rate of about 0.1077% per processor.

Nodes	Procs	OpenMP (sec)	MPI (sec)	Total	Parallel Efficiency	Speedup
1	1	721.921	0.189	722.110	1.0	1.0
1	8	92.979	0.210	93.189	0.969	7.749
2	16	46.490	0.530	47.020	0.960	15.357
4	32	23.606	0.630	24.236	0.931	29.795
8	64	11.874	0.970	12.844	0.878	56.219
16	128	5.964	1.390	7.354	0.767	98.191
32	256	3.175	1.440	4.615	0.611	156.470
64	512	1.701	1.609	3.31	0.426	218.095

Table 5.2: Parallel performance results for 2 million animals.

At larger scale, beginning with 10 million animals, the observed performance is quite different. Most noteably, in Figure 5.6, a sharp degradation in parallel efficiency is observed between 1 and 8 processors indicating a degradation in shared memory performance. This degradation is compensated for by an improvement in MPI performance as seen by the "flattening" of the slope over the interval $8-512$. The asymptotic MPI performance appears at or slightly before 10 million animals, as successively larger datasets also show nearly the same degradation rate of about 0.04% per processor.

For 10 million animals the increase in MPI performance offsets the degradation in shared memory performance at around 256 processors and corresponds to the intersection of the two parallel efficiency curves in Figure 5.6. When using more than 256 processors, the algorithm processes 10 million animals more efficiently than it does 2 million animals. For 10 million animals utilizing all 512 processors, a speedup of nearly 252 is observed corresponding to a 50% parallel efficiency. This is the peak observed parallel efficiency using all 512 processors. For datasets larger than 10 million animals, shared memory performance declines sharply in between 10

and 20 million animals, and then appears to slowly approach a steady-state degradation in parallel efficiency of 45%.

When executed at largest scale, 100 million animals, and utilizing all 512 processors, a speedup of nearly 172 is observed corresponding to a parallel efficiency of 33.5%. At this scale, the hybrid algorithm requires nearly a TByte of memory and stresses the full system resources; memories, caches, processors and network. The hybrid model requires the full utilization of all system resources to reduce the tracing time from 17 hours down to 6 minutes. The observed speedup is analogous to the difference between walking at a brisk pace of 5 miles an hour and traveling in a supersonic jet at a speed of over mach 1.

5.3.2 Analysis

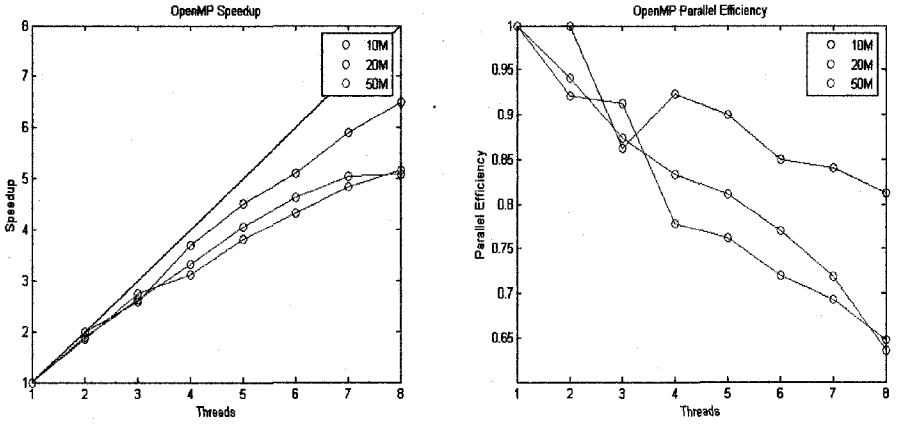

Figure 5.7: Speedup (left) and parallel efficiency (right) within an SMP node. For large datasets ($\geq$ 10 million animals), large strides increase main memory traffic. OpenMP threads compete for shared memory bandwidth and performance degrades. Total parallel performance is bound by this bottleneck.

Efficiency is determined by shared memory parallel performance. Within a node, the performance decreases as the datasets grow in size. In order to

better understand this performance bottleneck, refer back to Figure 2.1 in Chapter 2 which details the POWER5 microchip architecture. Each chip consists of a single processor core and a cache hierarchy consisting of a 2 MByte on-chip L2 cache, and a 36 MByte off-chip L3 cache. The L3 cache resides on the processor side of the fabric, thus L3 traffic does not increase traffic on inter-chip buses.

In Figure 5.7 the effects of shared memory contention can be seen in both the parallel speedup (left) and the parallel efficiency (right) for 10, 20 and 50 million animals. The small (2 million animal) dataset is small enough to make efficient use of the large L3 cache; memory contention is not observed and shared memory performance is excellent. Starting around 10 million animals, datasets become so large that cache hierarchies can no longer be efficiently utilized resulting in more memory traffic. Shared memory performance suffers as the number of threads increases since the main memory data path is shared among all threads. These observations imply that as the amount of memory used within a node increases, per processor performance decreases in proportion to the number of active threads.

5.4 Conclusions

In conclusion, a hybrid OpenMP/MPI disease tracing algorithm has been designed and mapped in a way which takes maximal advantage of the target architecture. This programming model has been demonstrated to facilitate efficient use of and access to system data resulting in high-performance. Observed parallel timings are excellent, 100 million animals can be processed in just over 6 minutes on 512 processors resulting in a speedup of 171, corresponding to a parallel efficiency of 33.5%.

At small-scale, shared memory performance is nearly optimal and performance degradation is due primarily to MPI overhead. As the population size increases, MPI performance improves and approaches a steady-state performance degradation rate of about 0.04% per processor. As problem size increases, the primary performance bottleneck is the shared memory parallel performance which degrades precipitously at 10 million animals due to memory bandwidth contention inherent in the L3 cache design.

High-performance is achieved and requires the full resources of the target architecture. Disease tracing is a memory intensive application and an SMP supercomputer with low-latency shared memory nodes interconnected via a high bandwidth switch is demonstrated to be an ideal candidate architecture to solve this problem on when coupled with a hybrid parallel programming model. The primamry performance bottleneck is observed to be shared memory contention which increases with problem size. On a single node, at large-scale only about four threads can be used highly efficiently. The performance data collected, when viewed as a whole, suggests that parallel performance can be greatly improved by doubling the number of nodes (and hence doubling the available memory) and halving the number of processors within a node.

Chapter 6

CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

A summary of the major conclusions of this work follows.

- A traceability simulation has been designed and implemented. A Monte Carlo process is used to produce large, NAIS-compliant mock datasets. The Monte Carlo simulation employs a new, fast, portable, parallel LFG random number generator and has been implemented on an SMP supercomputer. Animal populations ranging in size from 2 million to 100 million animals have been created for subsequent traceability scaling studies.

- A disease tracing algorithm has been mapped onto a large-scale, shared memory SMP supercomputer whose nodes are interconnected via a high-bandwidth switch. Based on insights gained via empirical machine benchmarking, the algorithm is mapped onto the target architecture in a way which takes maximal advantage of the memory hierarchy. It is found that an SMP architecture with large, shared memory nodes is well suited to the disease tracing problem when coupled with a hybrid parallel programming model.

- Dataset processing requirements depend upon the population size, premises size distribution, and transaction number distribution. In general, increasing the number of large-sized premises (e.g. very large feedlot premises) increases serial execution time. The single processor execution time is highly sensitive to changes in the premises size distributions.

- The current implementation is capable of accomplishing traceback on 100 million animals in under seven minutes when utilizing all 512 available processors. The same dataset, when processed on a single processor, takes over seventeen hours to process. This translates to an achievable speedup of 171 corresponding to a 30% parallel efficiency.

- Sequential execution time is dominated by the traceback phase, a nested-loop algorithm. In particular, the entire hybrid program, Algorithm 8, can be executed in about seventeen hours and forty-five minutes on a single processor. The traceback phase requires about seventeen hours. Of the remaining forty minutes, thirty minutes are used for I/O and about ten minutes are required to precompute the data structures required for efficient search and retrieval.

- When utilizing all 512 processors, the total execution time of the hybrid program (Algorithm 8) is dominated by I/O time which takes about 30 minutes for 100 million animals.

- Achieving high-performance when processing a National dataset (100 million animals) requires both a very large amount of memory and very fast access to that memory. Thus rapid traceback necessitates a

large-scale supercomputing platform. In this implementation, memory use scales as $O(nnodes * na)$ and na is bound by the amount of available memory on a single node and not the total aggregate system memory.

- A hybrid OpenMP/MPI parallel programing model is a highly effective programming paradigm that maps well onto an SMP supercomputer consisting of several shared memory nodes. The hybrid model facilitates both effective use of and efficient access to memory.

- For small datasets, parallel performance degradation is due almost exclusively to inter-node synchronization overhead (MPI), however the degradation rate is small, about 0.1077% per processor.

- MPI performances improves for large datasets, this is due to the high-bandwidth HPS switch. For populations larger than 10 million animals, performance degradation due to MPI overhead approaches a steady rate of about 0.04% per processor.

- The opposite is true of shared memory performance which is optimal for small datasets. A precipitous degradation occurs for large datasets, starting around 10 million animals. Shared memory degradation is the primary performance bottleneck, accounting for a 50% decrease in overall parallel performance for large datasets. Because of this, performance is bound by the performance within a single shared-memory node.

- Shared memory degradation can be explained by inefficient use of the large L3 cache. As the datasets grow in size, so too does the

average access stride per retrieval. Larger strides decrease the effective size of the cache hierarchy which leads to an increase in memory traffic. While increasing the number of OpenMP threads within a node decreases per processor workload, when both shared-data and access strides are large, average per processor performance decreases in proportion to the number of active threads computing.

- There are two ways to add processors on this architecture; processor cores can be added to computational nodes, or more computational nodes can be added by interconnecting them to the HPS switch. The hybrid OpenMP/MPI algorithm scales very well across nodes and scales poorly within a node. Significantly higher performance can be attained if the number of nodes is doubled and the number of processors within a node is halved.

6.2 Recommendations

- For even larger problems, a parallel model which distributes the data structures should be developed and implemented. Based on current results, this should improve shared memory performance. Decreasing the amount of memory used on each node will increase performance. Partitioning the data tasks amongst many computational nodes, however, could introduce load-balancing issues and fine-grained inter-nodal communication not present in the current implementation.

- A cluster of large, shared memory SMP nodes interconnected via a high-bandwidth switch is well suited to disease tracing. The current implementation should be ported to several different additional HPC

architectures. A study comparing and contrasting the performance of the current implementation on various HPC platforms with various processors, interconnects, and cache hierarchies should be conducted.

- A hybrid OpenMP/MPI programming model is well suited to this problem, but other parallel programming models should be explored also. In particular, the tracing algorithm should be implemented with a Partitioned Global Address Space (PGAS) programming model and implemented on a large, shared memory machine. In this model, portions of the shared memory space have an affinity for a particular thread, thereby exploiting locality of references.

- Robustness to changes in dataset parameters should be further studied. It was observed throughout the course of many simulations that small changes to the premises size distribution can lead to large changes in execution time.

- A study should be conducted to determine the effects of reduced participation in the NAIS. It is currently estimated that only 30% of producers have registered their premises and herds. This simulation could be used to empirically discover the critical mass necessary to contain an outbreak within a given confidence level.

- In this research it is assumed that the entire dataset is centrally located in storage. Due to various political and commercial interests this assumption may not be satisfied. The effects of distributing the dataset over a large geographic area should be studied by quantifying and categorizing asynchronous communication volume among retrieval and data tasks.

- When all processors are utilized, execution time is dominated by slow I/O, in particular reading the datasets into memory. MPI-IO should be explored as a viable way to address this overhead.

Bibliography

[1] A. Aho, J. Hopcroft, J. Ullman *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[2] G.S. Amalsi, A. Gottlieb *Highly Parallel Computing*, (2nd edition), Benjamin/Cummings, 1994.

[3] S.L. Anderson *Random Number Generators on Vector Supercomputers and Other Advanced Architectures*, SIAM Review, vol. 32(2), pp.221-251, 1990.

[4] T. Anderson, D. Culler, D. Patterson, et al. *A Case for Networks of Workstations: NOW*, IEEE Micro, 1995.

[5] M. Bane, R. Keller, M. Pettipher, I. Smith *A Comparision of MPI and OpenMP Implementations of a Finite Element Analysis Code*, In Proceedings of Cray User Group Summit (CUG 2000), May 22-26, 2000.

[6] D.J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer, *BEOWULF: A Parallel Workstation for Scientific Computing*, Proceedings of International Conference on Parallel Processing, 1995.

[7] R.P. Brent *Uniform Random Number Generators for Supercomputers*, Proceedings Fifth Australian Supercomputer Conference, SASC Organizing Committee, 1992, pp. 95-104.

[8] R.P. Brent *On the Periods of Generalized Fibonacci Recurrences*, in the Press, Math. Comput. (1994).

[9] P.J. Burns, J.S. Ladd *Implementation of a Fast, Portable, Parallel Random Number Generator*, In preparation 2008.

[10] P.J. Burns, D.V. Pryor *Surface Radiative Transport at Large-Scale via Monte Carlo*, Annual Review of Heat Transfer, 9, C.L. Tien, ed., pp.79-158, Begell House, New York, 1998.

[11] I.J. Bush, C.J. Noble, R.J. Allan *Mixed OpenMP and MPI for Parallel Fortran Applications*, In European Workshop on OpenMP 2000, Edinburgh, UK, 2000.

[12] F. Cappello, D. Etiemble *MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks* In Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Dallas, TX, 2000. IEEE Computer Society Press.

[13] F. Cappello, O. Richard, D. Etiemble *Investigating the performance of two programming models for clusters of SMP PCs*, In IEEE HPCA6, 2000.

[14] S.A. Cuccaro, M. Mascagni, D.V. Pryor *Techniques for Testing the Quality of Parallel Pseudorandom Number Generators*, Proc. of the 7th SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, 1995.

[15] J.L. Devore *Probablity and Statistics for Engineering and the Sciences fourth edition*, ITP Publishing, 1995.

[16] S. Dong, G.E. Karniadakis *Dual-Level Parallelism for Deterministic and Stochastic CFD Problems*, In Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Baltimore, MD, USA, 2002. IEEE Computer Society Press.

[17] N. Drosinos, N. Koziris *Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters*, IPDPS, p. 15a, 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Papers, 2004.

[18] A.M. Ferrenberg, D.P. Landau, Y.J. Wong *Monte Carlo simulations: hidden errors from "good" random number generators*, Phys. Rev. Let. 69 (1992) 3382-3384.

[19] M. Feyereisen, R. Kendall *An efficient implementation of the Direct-SCF algorithm on parallel computer architectures*, Theoretica Chimica Acta, 84:289-299, 1993.

[20] I. Foster *Designing and Building Parallel Programs Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Publishing, 1995.

[21] S. Goedecker, A. Hoisie *Performance Optimization of Numerically Intensive Codes (Software, Environments and Tools)*, SIAM Publishing, 2001.

[22] P. Grassberger *On Correlations in 'good' random number generators*, Phys. Rev. Lett. A 181 (1) (1993) 43-46.

[23] C. Grassel *Blended programming: MPI and OpenMP*, T.J. Watson Research Center presentations, IBM 1999.

[24] J. Handy *The Cache Memory Book*, Morgan Kaufmann Publishers, ISBN 0123229804, 1997.

[25] Y. He, C.H.Q. Ding *MPI and OpenMP Paradigms on Clusters of SMP Architecture: The Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition*, In Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Baltimore, MD, USA, 2002. IEEE Computer Society Press.

[26] D.S. Henty *Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling*, In Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, page 10,Dallas, Texas, 2000.

[27] P. Lanucara, S. Rovida *Conjugate-gradients algorithms: An MPI-OpenMP implementation on distributed shared memory systems*, In First European Workshop on OpenMP, pages 76-78, Lund, Sweden, 1999.

[28] R.D. Loft, S.J. Thomas, J.M. Dennis *Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models*, In Proceeding of the 2001 ACM/IEEE Conference on Supercomputing, pages 18-18, Denver, CO, 2001.

[29] M.H. Kalos, P.A. Whitlock *Monte Carlo Methods Volume I: Basics*, Wiley-Interscience Publication, 1986.

[30] D. Klepacki *Mixed-mode programming*, T.J. Watson Research Center Presentations, IBM 1999.

[31] D.E. Knuth *The Art of Computer Programming, Vol.2 Seminumerical Algorithms, 2nd ed.*, Addison-Wesley, Reading, Massachusetts (1981).

[32] D.E. Knuth *The Art of Computer Programming, Vol.3 Sorting and Searching, 2nd ed.*, Addison-Wesley, Reading, Massachusetts, 1981.

[33] G. Krawezik, F. Cappello *Performance Comparisons of MPI and Three OpenMP Programming Styles on Shared Memory Multi-Processors*, IN ACM SPAA 2003, Sna Diego, CA, 2003.

[34] V. Kumar, A. Grama, A. Gupta, G. Karypis *Introduction to Parallel Computing Design and Analysis of Algorithms*, Benjamin/Cummings Publishing, 1994.

[35] *Parallel Performance Study of Monte Carlo Photon Transport Code on Shared, Distributed, and Distributed-Shared-MemoryArchitectures*, Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), P. 93-, 2000.

[36] J.D. Maltby, B.T. Kornblum *MONT3E: A Monte Carlo Electron Heat Transfer Code*, Proceedings of the 1990 Conference on Supercomputing, 1990.

[37] J.D. Maltby *Analysis of Electron Heat Transfer via Monte Carlo Simulation*, PhD Dissertation, Department of Mechanical Engineering, Colorado State University, 1990.

[38] G. Marsaglia *A Current View of Random Number Generators*, in Computer Science and Statistics: Proceedings of the XVIth Symposium on the Interface, 1985, pp. 3-10.

[39] M. Mascagni, S.A. Cuccaro, D.V. Pryor, M.L. Robinson *A Fast, High Quality, and Reproducible Parallel Lagged-Fibonacci Pseudorandom Number Generator*, Journal of Computational Physics, 1994.

[40] M. Mascagni *Some Methods of Parallel Pseudorandom Number Generation*,in Proceedings of the IMA Workshop on Algorithms for Parallel Processing, Springer-Verlag, 1997.

[41] J.M. May, B.R. Supinski *Experiences with mixed MPI and threaded programming models*, Center for Applied Scientific Computing, presentation at the IBM Advanced Computing Technology Center SP Scientific Applications and Optimizations Meeting at the San Diego Supercomputing Center, 1999.

[42] MPI-Forum, *MPI Forum*, 1999. http://www.mpi-forum.org.

[43] NERSC Web documentation *NERSC*, LBNL, http://www.nersc.gov/nusers/systems/bassi/more.php

[44] NIDT 2003 *United States Animal Identification Plan*, version 4.1. National Identification Development Team. http://usaip.info/USAIP4.1.pdf.

[45] OpenMP *The OpenMP ARB*, http://www.openmp.org/

[46] W.F. Press, B.P. Flannery, S.A. Tuekolsky and W.T. Vetterling *Numerical Recipes in FORTRAN*, Cambridge University Press, pp. 191-225, 1988.

[47] B.V. Protopopov, A. Skjellum *Shared-memory communication approaches for an MPI message-passing library*, Concurency: Practices and Experiences, 12:799-820, 2000.

[48] B.V. Protopopov, A. Skjellum *A Multi-Threaded Message Passing Interface (MPI) Architecture: Performance and Program Issues*, JPDC, 2001.

[49] D.V. Pryor, S.A. Cuccaro, M. Mascagni, M.L. Robinson *Implementation and Usage of a Portable and Reproducible Parallel Pseudorandom Number Generator*, 1994.

[50] R. Rabenseifner, G. Wellein *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures*, International Journal on High Performance Computing Applications, 17(1):49-62, 2003.

[51] M. Resch, B. Sander *A Comparision of OpenMP and MPI for the Parallel CFD Test Case*, In Proceedings of the first European Workshop on OpenMP, Lund, Sweden, p. 71-75, 1999.

[52] J. Scanga, T. Hoffman, J. Picanso, S.V. Rajopadhye, D.G. Kim, A. Gupta, R. Forbes, J.S. Ladd, P.J. Burns *Development of Computational Models for the Purpose of Conducting Individual Livestock and Premise Traceback Investigations Utilizing NAIS Compliant Data*, Journal of Animal Science, 2006.

[53] R. Sedgewick *Algorithms in C*, Addision-Wesley, Reading, MA, 1990.

[54] A.E. Segarra and J.M. Rawson *Mad Cow Disease: Agricultural Issues*, U.S. Department of State, http://fpc.state.gov/fpc/6121.htm

[55] H. Shan, J.P. Singh, L. Oliker, R. Biswas *Message passing and shared address space prallelism on an SMP cluster*, Parallel Computing, submitted.

[56] K. Shen, H.Tang, and T. Yang *Adaptive Two-Level Thread Management for Fast Execution on Shared Memory Machine,* In Proceedings of ACM/IEEE Supercomputing '99, New York, November 1999. ACM/IEEE.

[57] B. Sinharoy, R.N. Kalla, J.M. Tendler, R.J. Eickenmeyer, J.B. Joyner *POWER5 System Microarchitecture*, IBM Journal of Research and Development, Vol. 49, NO. 45, July/September 2005.

[58] L.A. Smith *Mixed Mode MPI/OpenMP Programming*, Technical Report (UKHEC 2000), http://www.ukhec.ac.uk/publications/tw/mixed.pdf,

[59] L.A. Smith, P. Kent *Development and performance of a mixed OpenMP/MPI quantum Monte Carlo code*, Concurrency: Practice and Experience, 12:1121-1129, 2000.

[60] A. Srinivasan, M. Mascagni, D. Ceperley *Testing Parallel Random Number Generators*, Parallel Computing29, 2003, pp. 69-94 Physica D, 61:260, 1992.

[61] H. Tang, K. Shen, T. Yang *Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines*, ACM Transactions on Programming Languages and Systems, 2000.

[62] H. Tang, T. Yang *Optimizing Threaded MPI Execution on SMP Clusters*, ICS, P. 381-392, 2001.

[63] A. Tucker *Applied Combinatorics*,John Wiley and Sons, 1995.

[64] USDA. 2006b. *National Agricultural Statistical Service-Quick Stats*, USDA, National Agricultural Statistical Services. http://www.nass.usda.gov/index.asp.

[65] USDA. 2008 *National Agricultural Statistical Service*, USDA, National Agricultural Statistical Services, http://usda.mannlib.cornell.edu/usda/current/Catt/Catt-07-25-2008.pdf

[66] USDA. 2007 *United States Department of Agriculture Fact Sheet*, USDA, http://animalid.aphis.usda.gov/

[67] USDA. December 2007 *United States Department of Agriculture Fact Sheet*, USDA, http://animalid.aphis.usda.gov/

[68] USDA-APHIS, 2008 *USDA-APHIS*, USDA-APHIS, http://animalid.aphis.usda.gov/nais/why/bse.shtml#bse

[69] R. Van der Pass *Memory Hierarchy in Cache Based Systems*, Sun Blueprints, 2002.

[70] IBM *XL Fortran Enterprise Edition for AIX - User's Guide*, Version 9.1, http://www.nersc.gov/vendor_docs/ibm/pdf/xlf_9.1.0.2_ug.pdf

[71] C.N. Zeeb *Performance and Accuracy Enhancements of Radiative Heat Transfer Modeling via Monte Carlo*, PhD Dissertation, Department of Mechanical Engineering, Colorado State University, 2002.

Appendix A

STATISTICAL TESTING PROCEDURES

In this section a rigorous staticstical framework is built, within which, empirical tests on the performance and quality of the parallel ALFG are performed. The theory of statistics provides quantitative measures for randomness. Since an RNG is supposed to produce a sequence of independent, identically distributed (i.i.d.) random variables X_1, X_2, X_3,..., any statistic that is computed with elements of the sequence may serve as a test, if its *distribution* is known. If the distribution of the statistic, computed using an RNG sequence, is sufficiently close to the expected distribution which assumes a sequence of i.i.d. random variables, then the RNG is said to be *good*. If it is far from the expected distribution, then the RNG is considered defective. In this section, statistical metrics are defined and a precise definition of close is provided. Once well defined, these metrics can be used to assess both the quality of the individual ALFG streams (testing for intra-stream correlation) as well as the effectiveness of the proposed parallel initialization (testing for inter-stream correlation).

A.1 The Kolmogorov-Smirnov Test

A general way to specify the distribution of a random variable X, be it discrete or continuous, is in terms of the distribution function $F(x)$

$$F(x) = p(x \leq X) \tag{A.1}$$

where $p(E)$ is the probability of event E.

If n independent observations of the random quantity X are made, thereby obtaining the values X_1, X_2, ..., X_n. Then the empirical distribution function (EDF) $F_n(x)$ is defined as

$$F_n(x) = \frac{number\ \ of\ \ X_1, \ldots, X_n\ \ that\ \ are \leq x}{n}. \tag{A.2}$$

If the observed data are actually distributed according to $F(x)$. Then as $n \rightarrow \infty$, $F_n(x)$ converges, in measure, to $F(x)$.

The Kolmogorov-Smirnov test (KS test) can be applied when $F(x)$ is continuous [31]. It is based on the difference between $F(x)$ and $F_n(x)$. A bad source of random numbers will produce EDFs which do not approximate $F(x)$ sufficiently well.

To make a KS test, the following statistics are formed:

$$K_n^+ = \sqrt{n}\ \ max(F_n(x) - F(x)), \quad -\infty \leq x \leq \infty; \tag{A.3}$$

$$K_n^- = \sqrt{n}\ \ max(F(x) - F_n(x)), \quad -\infty \leq x \leq \infty. \tag{A.4}$$

These values are distributed according to the Kolmogorov distribution, and may be obtained in a percentile lookup table, such as the one provided in [31], to determine if the values are significantly high or low.

A distinct advantage to the KS test is that it provides an exact distribution for any number of observations, n. A KS test may be employed reliably, even when the number of observations is small. This is in contrast to the χ^2 test, which will be introduced in the next section, which provides an asymptotic distribution that is only valid for a large number of observations.

A.2 The χ^2 Test

The χ^2 test (chi-square test) is one of the best known and most basic methods to compare two discrete probability distribution functions. In general, suppose that every observation of the random variable falls into one of k possible categories. Consider n independent observations of the random quantity X (independent means that the outcome of one observation has absolutely no effect on the outcome of any of the others). Let p_s be the probability that some observation falls into category s. Let Y_s be the observed number of trials that fall into category s. The expected number of trials which fall into category s is $p_s n$. With this, the chi-square statistic is defined as

$$V = \sum_{s=1}^{k} \frac{(Y_s - p_s n)^2}{p_s n}. \tag{A.5}$$

This statistic is distributed according to the chi-square distribution with $\nu = k - 1$ degrees of freedom. With distribution function

$$F_{\chi^2_\nu}(x) = \frac{\gamma(\nu/2, x/2)}{\Gamma(\nu/2)} \tag{A.6}$$

where

$$\gamma(a, x) = \int_0^x t^{a-1} e^{-t} dt \tag{A.7}$$

and

$$\Gamma(a) = \int_0^\infty t^{a-1} e^{-t} dt. \tag{A.8}$$

This test is applied whenever observations fall into a discrete set of categories. In other words, a chi-square test can be employed when the random variable in question is distributed according to a discrete probability distribution (as is the case when flipping coins or rolling die.) Discrete probability distributions give rise to discontinuous distributions in which

case the KS test is not applicable. As previously mentioned, the distribution in A.6 is an asymptotic result which is only valid for a large number of observations.

A more effective strategy for testing a random number sequence is to use the chi-square test in conjunction with the KS test [31]. To illustrate how this might be done, suppose 10 independent χ^2 tests have been made on different parts of a random sequence, so that values V_1, V_2,...,V_{10} have been obtained. It does not suffice to simply count the number of suspiciously large or small values. A better procedure is to compute the empirical distribution $F_{10}(x)$, perform a KS test, and obtain the statistics K_{10}^+, K_{10}^-. This gives a clearer picture of the results of the χ^2 test. It may be the case that individually, each value passes the chi-square test, yet collectively these observations are not at all correct (see [31] for an example of this).

Appendix B

PARALLEL PERFORMANCE RESULTS

B.1 Traceback Timing Results

Nodes	Procs	OpenMP (sec)	MPI (sec)	Total	Parallel Efficiency	Speedup
1	1	6,180.810	1.030	6,181.84	1.0	1.0
1	8	950.776	1.020	951.796	0.812	6.495
2	16	476.814	7.970	484.784	0.797	12.752
4	32	238.572	6.480	245.052	0.788	25.227
8	64	120.604	6.579	127.183	0.756	48.606
16	128	61.125	6.430	67.555	0.715	91.509
32	256	31.097	7.690	38.787	0.623	159.376
64	512	13.832	10.710	24.542	0.492	251.884

Table B.1: Parallel performance results for 10 million animals.

Nodes	Procs	OpenMP (sec)	MPI (sec)	Total	Parallel Efficiency	Speedup
1	1	13,398.617	1.960	13,400.577	1.0	1.0
1	8	2,683.292	1.940	2,685.232	0.624	4.990
2	16	1,349.73	4.280	1,354.010	0.625	10.000
4	32	675.448	7.410	682.858	0.613	19.624
8	64	339.075	12.910	351.985	0.595	38.071
16	128	172.840	13.060	185.900	0.563	72.084
32	256	88.474	15.140	103.614	0.505	129.331
64	512	45.729	17.690	63.419	0.412	211.301

Table B.2: Parallel performance results for 20 million animals.

Nodes	Procs	OpenMP (sec)	MPI (sec)	Total	Parallel Efficiency	Speedup
1	1	23,493.641	4.520	23,498.161	1.0	1.0
1	8	4,618.628	4.468	4,623.096	0.635	5.083
2	16	2,229.586	10.357	2,239.943	0.656	10.491
4	32	1,181.014	16.211	1,197.225	0.613	19.627
8	64	581.948	30.611	612.559	0.599	38.361
16	128	302.505	28.630	331.135	0.554	70.962
32	256	148.936	38.650	187.586	0.489	125.266
64	512	76.614	40.770	117.384	0.391	200.182

Table B.3: Parallel performance results for 50 million animals.

In this section, performance data for all datasets processed in Chapter 5 is presented. In particular, parallel timing and performance results for 10, 20, and 50 million animals are tabulated. The total parallel execution time is the sum of the OpenMP retrieval phase with the MPI scatter-gather phase. Both parallel efficiency and speedup are measured.

B.2 Shared-Memory Performance

In this section, shared-memory parallel performance results are presented. Only datasets of size 10, 20, and 50 million animals are considered. In these runs, the hybrid code is executed on a single SMP node and the number of threads is increased from one to eight. Since no MPI communication occurs parallel results measure only shared-memory performance.

The data contained in Tables B.4, B.5, B.6 is plotted in Figure 5.7 in Chapter 5. The data shows that shared-memory performance contributes to an overall 19% decreases in parallel efficiency. Shared memory degradation is negligible for 2 million animals and thus the results are not included herein. For both 20 million and 50 million animals, shared-memory degra-

dation contributes to an overall 36% decrease in parallel efficiency and is the dominant performance bottleneck.

Threads	Time (sec)	Parallel Efficiency	Speedup
1	6,180.810	1.0	1.0
2	3,088.763	1.0	2.0
3	2,388.315	0.863	2.588
4	1,673.392	0.923	3.694
5	1,373.035	0.900	4.502
6	1,210.621	0.851	5.106
7	1,047.543	0.843	5.900
8	950.776	0.813	6.501

Table B.4: Shared-memory parallel performance results for 10 million animals.

Threads	Time (sec)	Parallel Efficiency	Speedup
1	14,235.472	1.0	1.0
2	7,730.157	0.921	1.841
3	5,195.155	0.913	2.740
4	4,572.177	0.778	3.113
5	3,738.290	0.762	3.808
6	3,293.327	0.720	4.322
7	2,936.956	0.692	4.847
8	2,751.069	0.647	5.175

Table B.5: Shared-memory parallel performance results for 20 million animals.

Threads	Time (sec)	Parallel Efficiency	Speedup
1	23,493.641	1.0	1.0
2	12,491.451	0.940	1.881
3	8,954.275	0.875	2.624
4	7,051.483	0.833	3.332
5	5,789.175	0.812	4.058
6	5,080.830	0.771	4.624
7	4,667.434	0.719	5.033
8	4,618.628	0.636	5.087

Table B.6: Shared-memory parallel performance results for 50 million animals.