CLUE: A CLUSTER EVALUATION TOOL

Brandon S. Parker, B.S.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2006

APPROVED:

Armin R. Mikler, Major Professor
Steve Tate, Committee Member
Tom Jacob, Committee Member
Phil Sweany, Committee Member
Krishna Kavi, Chair of the Department of Computer Science and Engineering
Oscar Garcia, Dean of the College of Engineering
Sandra L. Terrell, Dean of the Robert B. Toulouse School of Graduate Studies
Modern high performance computing is dependent on parallel processing systems. Most current benchmarks reveal only the high level computational throughput metrics, which may be sufficient for single processor systems, but can lead to a misrepresentation of true system capability for parallel systems. A new benchmark is therefore proposed. CLUE (Cluster Evaluator) uses a cellular automata algorithm to evaluate the scalability of parallel processing machines. The benchmark also uses algorithmic variations to evaluate individual system components’ impact on the overall serial fraction and efficiency.

CLUE is not a replacement for other performance-centric benchmarks, but rather shows the scalability of a system and provides metrics to reveal where one can improve overall performance. CLUE is a new benchmark which demonstrates a better comparison among different parallel systems than existing benchmarks and can diagnose where a particular parallel system can be optimized.
Copyright 2006

by

Brandon S. Parker
ACKNOWLEDGMENTS

I would like to thank the numerous individuals that supported me in my endeavors to obtain this master’s degree. Particularly, I would like to thank Dr. Mikler for his encouragement, insight, and overall enthusiasm throughout my college studies. I would like to thank Dr. Tate for sharing his knowledge and perspective on numerous computational subjects, not the least of which include security and algorithms. Thanks also to Dr. Jacob, Dr. Sweany, and the various other professors at UNT who have contributed to my understanding and obsession with all things computer related.

I am grateful to my parents for their support and encouragement throughout my schooling. I would like to thank my mother for her grammar and spell checking, as well as leveraging her librarian skills to find papers and journals which alluded me. Thanks also to my co-workers, since I doubt I could have completed my graduate work as quickly if it were not for their flexibility and understanding either. I would like to especially thank my friend and future wife, Cheryl-Annette, for her support, sanity, creativity, and patience throughout my rants, brainstorms, and writing/programming marathons.
CONTENTS

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER 1. INTRODUCTION 1
  1.1. Cluster Computing 1
  1.2. Evaluating Parallel Systems 3
  1.3. Overview 6

CHAPTER 2. CLUSTER BENCHMARKING 8
  2.1. Parallel Scalability 9
  2.2. Bottlenecks to Performance 11
  2.3. Modern Scalability Analysis 13
  2.4. Other Benchmarks 15
    2.4.1. LinPack 15
    2.4.2. HINT 16
    2.4.3. NetPIPE 18
  2.5. Benchmarking Scalability and Efficiency 18

CHAPTER 3. BENCHMARK DESIGN 21
  3.1. Using Cellular Automata as a Benchmark 21
  3.2. Designing a Good Benchmark 25
  3.3. Scalability Analysis 31
  3.4. Systematic Testing and Comparative Analysis 31
3.5. Reporting 33

CHAPTER 4. RESULTS AND ANALYSIS 35
4.1. Result Metrics 35
4.2. Performance Analysis 39
4.3. Scalability Analysis 42
4.4. Summary of Results 44

CHAPTER 5. SUMMARY AND CONCLUSION 46
5.1. Summary 46
5.2. Future Work 46
5.2.1. Expanding Environments 46
5.2.2. Expanding Tests 47
5.2.3. Expanding Analysis 47
5.2.4. Further Testing 48
5.2.5. User Interface 48
5.3. Conclusion 49

BIBLIOGRAPHY 50
LIST OF TABLES

4.1 Cluster tested configurations. 36
4.2 CLUE results by processor count for the base line variation. 37
4.3 CLUE results by variation type using eight nodes. 38
LIST OF FIGURES

3.1 A 16x16 torus cellular automata grid layout.  23
3.2 A cellular automata grid spread across 4 nodes.  26
3.3 System factors for comparative analysis.  27
3.4 CLUE Perl wrapper algorithm.  28
3.5 CLUE core benchmarking algorithm.  30

4.1 Five basic network topologies.  40
4.2 ACT/s by variation.  41
4.3 Efficiency by variation.  42
4.4 Serial fraction by variation.  43
4.5 Efficiency by nodes.  44
4.6 Serial fraction by nodes.  45
CHAPTER 1

INTRODUCTION

Scientific and other data intensive applications need more computing power than in previous years. There exists a distinct limit to the computational power of a single processor, so parallel processing is a popular solution for faster and larger applications. However, the performance gain achieved through parallel processing is limited by numerous factors. The majority of applications that are not designed specifically for parallel processing cannot increase performance with additional available processors. Even if an application is redesigned to make use of more processors, the performance is still limited by algorithm, system, and hardware factors. Despite a programmer’s best efforts, a small portion of the application cannot be parallelized. At a minimum, the system must load the code into memory and initialize the necessary system memory for data. This limit involving the serial overhead of an application is known as Amdahl’s law [2] after a paper presented by Gene Amdahl. He and others also point out the importance of system and hardware impact in the overall performance of a parallel system. Under normal conditions, it is impossible to achieve a perfect linear speedup in parallel processing due to the overhead of system messaging, device and network latency, and data synchronization. Despite these limits, parallel processing has been shown to be effective for many applications [7],[11].

1.1. Cluster Computing

Recently, massive parallel processing has been achieved in the form of cluster computing – the interconnecting numerous independent computing systems to divide a computing load. Since most computational clusters utilize consumer-grade computer systems, clusters have an attractive cost to performance ratio. The distinct separation of the processors and memory in the cluster however, leads to potentially larger performance inefficiencies than processor-array or symmetric multi-processing (SMP) machines. These limits to peak performance,
primarily caused by the network latency coupled with the separation of memory, are minimized through the use of high bandwidth and low latency networks, such as fast fiber optic networks. The cluster architecture has a distinct advantage over SMP or processor-array architectures, however, in that each processor has its own memory unit. This increases the overall memory size of the system and allows optimization by dividing the data set into distinct segments among the memory and processor units [10]. It has been shown that many applications have a tendency toward spatially and temporally arranged data [12]. An application’s performance is generally limited more by the lack of available memory on a system than by the raw processor execution speed [18]. Thus, by spreading the data over the cluster nodes and localizing the data to the corresponding local node processor, a performance increase can be achieved compared to referencing memory across the larger system [4]. This can also increase performance by eliminating swapping and paging associated with memory limits.

Cluster computing has uses outside high performance computing. Cluster architectures are used in environments where automatic fail-over of a service is required or where a service could benefit from load balancing, such as a server farm [3]. Although these two cluster architecture implementations may improve performance, their primary concern is maintaining service availability rather than processing a single application quickly via a distributed paradigm. This thesis focuses on the high performance cluster implementations and other multi-processor architectures. The performance of fail-over and load balancing service clusters are beyond the scope of this work.

Many of the world’s fastest computers are cluster architectures. A list of the world’s top supercomputers is maintained at http://www.top500.org. The entries there are evaluated and ranked using a benchmark application known as LinPack. LinPack performs a series of linear algebra subroutines to calculate performance metrics for the given machine or system. The results are given in the form of GFLOP/s (billions of floating point operations per second) according to a given data size. These metrics are then used by manufacturers, programmers, and system users to evaluate systems and better understand the current state of high performance computing [4]. While the metric of floating point calculation throughput
is important to many applications, a single metric cannot reveal the true performance of a given system [4]. A measure of speedup and system efficiency for scalability analysis is important in fully understanding the performance benefits of a cluster, especially in relation to the cost of the cluster size [15],[11].

1.2. Evaluating Parallel Systems

Speedup is measured by executing an application on an increasing number of parallel processors and evaluating the execution time [11]. In ideal conditions, every time the number of processors doubles, the execution time should decrease by a factor of two. As Amdahl points out though, this speedup is unachievable due to system overhead and hardware latency [2]. Gustafson was able to achieve a near perfect speedup, thus claiming the formula known as Amdahl’s law was inaccurate [7]. As Shi later pointed out [15], Gustafson’s re-evaluation was simply another formulation of Amdahl’s conclusions. What is known now as Amdahl’s law, and Gustafson’s modified version (known as Gustafson’s law) both focus on how an application can be divided among parallel processors, taking into account the portions of the application that can and cannot be optimized or sped up through distribution among the parallel processors. Amdahl’s law as it is known today was formulated by Ware in summary of earlier parallel efficiency studies [8],[20]. This formulation states that the speedup $s$ of an application with serial fraction $f$, parallel fraction $(1 - f)$, and $P$ number of processors is given by the formula [7],[21],[11],[12],[20]:

$$s = \frac{1}{f + \frac{(1-f)}{P}}$$

According to this formulation, speedup of an application is limited by the serial portion of the application as it is executed on multiple processors. From this programmatic analysis it becomes apparent that the effect of diminishing returns establishes a limit on the number of processors an application can utilize in a cost effective manner. Years after Amdahl published his paper and the formula which carries his name became known, Gustafson took another approach and re-evaluated formula (1). Gustafson argues formula (1) is impractical because
it assumes the problem size remains constant, whereas most application users keep execution
time constant and increase the problem size or computational precision when given additional
processor resources. Gustafson’s reformulation, which measures scaled speedup, assumes the
problem size \( k \) varies linearly with the number of processors \( P \). Assuming a serial fraction \( f \),
the new formula states [7]:

\[
(2) \quad s_k = f + (1 - f)P
\]

This new evaluation gives a stronger initial confidence in the use of parallel processing. However, as Shi demonstrates, formula (2) and formula (1) are really the same formula
approaching the problem from different angles. Shi points out that both formulas fail to
account for many important factors inhibiting parallel processing efficiency [15]. Both for-
mulas assume one can properly identify the serial and parallel proportions of the application.
This identification is non-trivial due to compiler optimizations and latencies involved in the
hardware implementation of the parallel processing system. As Amdahl stated in his original
1967 paper, parallel processing speed may be inhibited by uneven data distribution, signal
propagation delays (i.e. network overhead), and the speed at which a processor can access
data which may be dependent on another processor’s calculation [2]. The state of computer
hardware and architecture today is obviously different than it was in 1967, but Amdahl was
correct in his assessment. Shi and others formalize the potential points in a parallel system
on which efficiency depends. These include [2],[4],[15],[7]:

- Single-processor architecture and processing power
- Network speed and latency
- I/O subsystem of each node
- Problem size
- Problem input distribution
- Parallel percentage of the application code
- Parallel algorithm used
• Operating System overhead
• Language and compiler characteristics
• Application load time
• Memory initialization

All these factors can contribute to an observed serial percentage which is higher than the assumed serial percentage from the application code alone. This has led to the concept of measuring the actual execution time of an application on a varying number of processors to determine the serial percentage from the experimental comparisons, rather than predicting the speedup from a theoretical formula alone. The experimentally determined serial fraction is arguably more useful as it yields a more comprehensive picture of the efficiency, price/performance ratio, and other metrics [18]. The particular limitations to performance are not readily determined as the serial fraction is the aggregate cost of the above list. This has led to more simplistic formulas for execution time, where $T_r$ is time spent on serial portions, $T_{1-r}$ is time spent on parallel portions of the application, and the application utilizing $P$ processors [15]:

$$T(P) = T_r(P) + T_{1-r}(P)$$

and thus speedup $s$ is defined as:

$$s = \frac{T(1)}{T(P)}$$

Examining the experimentally determined serial percentage, speedup, and efficiency can give a better understanding of how to approach the development of parallel processing applications and help to identify bottlenecks in parallel processor systems [11]. Comparing a system’s performance to other systems, or comparing an application’s performance to variations of the application, can help developers and researchers fine tune systems and applications to gain better performance. Such comparative performance analysis, or benchmarking, when
published can help potential users, system purchasers, and programmers make decisions regarding system usage and acquisition. Benchmarks also help researchers and others in the high performance computing market to gain a better understanding of the overall state of high performance computing and to better collaborate [4]. Many benchmarks in existence, specifically those used for quick system comparisons and marketing purposes, rely on single metric results such as peak number of floating point operations per second (GFLOP/s). No single metric, however, can adequately describe total system performance capabilities [4].

While floating point operations are historically more computationally expensive than integer based instructions, and even though many high performance and research applications require floating point calculations, this single metric does not reveal a parallel processing system’s true effectiveness executing parallel applications. In order to gain a full understanding of a parallel system’s performance, a benchmark is needed which gives metrics such as parallel efficiency and speedup, and uses algorithmic variations to identify system limitations. Shi made a call for such an application saying [15],

“What we need is a practical engineering tool that can help us identify performance critical factors for any algorithm and processing environment with systematic and practical steps.”

1.3. Overview

This paper details a new benchmark application design intended to fulfill such a requirement. The design uses the concepts of experimentally derived speedup, efficiency, and serial percentage to gather performance metrics of parallel processing systems. The design uses a simplistic cellular automata algorithm with fine tuned variations to test different aspects of the computing environment. Comparing the derived metrics of the algorithm variations allows the user to more accurately determine system efficiency and identify potential parallel processing bottlenecks.

The remainder of this paper demonstrates the validity and usefulness of such a benchmark. Chapter 2 further details the need for such a benchmark, the contributing and historical
research used, and the basic formulation and design of the benchmark for scalability analysis. Chapter 3 examines the algorithms and implementation aspects of the benchmark. Chapter 4 presents the empirical results from the tested benchmark and offer analysis and insight into what system details the benchmark can give to a user. Chapter 5 contains concluding remarks and a summary of the results.
CHAPTER 2

CLUSTER BENCHMARKING

While there are numerous benchmarks in existence, few are designed for multi-processor environments. Those benchmarks in existence for parallel environments focus on throughput rather than scalability. Most simply return a metric describing raw calculation speed, such as the number of floating point operations for a given data size compared to the execution time. A single metric, however, only shows how a system might perform under certain conditions and for certain applications [4]. A clearer and more complete picture emerges when testing various components and the scalability of the computational system. Cluster scalability analysis is usually performed by executing an application using a varying number of processors and then comparing execution times [11]. Gathering data for results in this manner can be time consuming and difficult.

To address these limitations and gain a better understanding of parallel systems, this thesis proposes a new benchmark application which examines a system’s scalability and efficiency. The application should utilize a simple, yet scalable algorithm for system performance analysis by executing it on a varying number of processing nodes and then repeating the process while carefully altering certain aspects of the application to test different system components. For instance, the application can perform purely integer-based computations during one execution, and then floating point calculations during a subsequent execution. The comparison of the two execution times would then indicate the cost of floating point computations in the algorithm. In another case, the application can alter the ordering of the processing nodes to determine the effects of node arrangement and network topology on performance. The application’s performance as the processor count increases indicates the system’s overall scalability and efficiency. The comparative analysis of the application variations then help to
determine possible performance bottlenecks and the ideal applications and data arrangements for the particular system.

2.1. Parallel Scalability

In his 1967 paper [2], Gene Amdahl discusses the potential pitfalls and performance hurdles to a multi-processor computing approach. He claims the overhead associated with interprocessor communication and data management becomes burdensome to performance. He also explains that an application’s divisibility is limited by the inherent serial portion associated with data management and other amortized costs [2]. Amdahl’s law (formula (1)) shows a pessimistic view of parallel processing effectiveness [7]. For instance, assume an application is ninety-nine percent parallelizable. Thus, \( f = .01 \) and \( (1 - f) = .99 \). Compared to executing the application on one processor, a small cluster of sixteen processors yields:

\[
s = \frac{1}{.01 + \frac{.99}{16}} \approx 13.913
\]

If the same application is executed on a moderate sized cluster of 512 processors, the speedup is:

\[
s = \frac{1}{.01 + \frac{.99}{512}} \approx 83.797
\]

A speedup of just over eighty-three fold using 512 processors is poor. At this rate, the cost to performance ratio of adding more nodes to the execution becomes high. Alan Karp and Horace Flatt help to bring the performance into better perspective by calculating the efficiency of a cluster through the comparison of the speedup to the number of nodes or processors used. Thus, the efficiency \((e)\) of a cluster using \( s \) as the speedup and \( P \) as the number of processors is described by the formula [11]:

\[
(5) \quad e = \frac{s}{P}
\]
When the ninety-nine percent parallizable application is executed with sixteen processors obtaining a speedup of 13.9 fold, the efficiency is:

\[ e = \frac{13.913}{16} \approx 0.869565 \approx 86.9\% \]

However, when \( P \) is increased to 512 processors with a speedup of approximately eighty-four times faster, the efficiency plummets to:

\[ e = \frac{83.797}{512} \approx 0.163666 \approx 16.4\% \]

This incredibly low outcome of efficiency may at first seem a detrimental blow to massively parallel system usability, but the formula derived from Amdahl’s work should not be haphazardly and universally applied to general parallel processing. John Gustafson published his own formula in 1988 and attested to parallel performance outcomes which seemed to break Amdahl’s law. At Sandia National Labs, Gustafson tested an application with a claimed serial percentage of 0.4% – 0.8% and a speedup of 1021-fold using 1024 processors. This far exceeds the expectations of Amdahl’s law. Gustafson, however, made a fundamental change in speedup analysis. Where in Amdahl’s law the problem size remained constant and execution time decreased with added processors, Gustafson varied the problem size to the number of processors, thus keeping the execution time constant. Using this variant of speedup analysis, Gustafson derived a new scaled speedup, which is formula (2) [7]. Using the same example as before, the scaled speedup of an application using Gustafson’s law with \( f = 1\% \) and 512 nodes is:

\[ s_k = 0.01 + 0.99 \times 512 = 506.89 \]

and the efficiency, using the above efficiency formula, would then be:

\[ e = \frac{506.89}{512} \approx 99.002\% \]
This is remarkably more optimistic in regards to parallel processing effectiveness. The only difference between the two speedup formulas is which to hold constant: execution time, or problem size (or computational complexity).

2.2. Bottlenecks to Performance

As Amdahl [2] and later Shi [15] and others [10],[18],[11],[4] state, there are several factors contributing to slowdowns in parallel processing. These factors include, but are not limited to:

- Single-processor architecture and processing power
  The power and capabilities of the individual processors used have a strong bearing on the overall computational effectiveness of the parallel system. Although calculations can be distributed, if the individual processors have weak floating point calculation units, small cache, or slow memory access, the overall performance on the parallel system suffers [4].

- Network speed and latency
  The more dependent a processor’s calculations are to the calculations of peer processors, the more communication amongst the processors can slow down the overall system performance. The network topology and physical network medium can affect the performance of the system. Some parallel applications favor a low latency network over one with high bandwidth. Other parallel applications require each node to communicate with only a small subset of peer nodes [10]. The geometrical arrangement of the processors in relationship to each other compared to the division and shape of the data can cause slowdown when the data is accessed or traversed in different orders or dimensions [2].

- I/O subsystem of each node
  Related to the network speed and latency and to disk and memory performance, the underlying subsystems of each node in a cluster can have grave impact on parallel processing performance. In SMP architectures, the underlying subsystems
can become overloaded by processor requests, where in a cluster environment, each
processor may have dedicated memory and system buses [15],[4].

- **Problem size**
  The overall complexity and size of the application has a direct impact on parallel
processing performance. If the problem size is too large, it can overburden the
system and require data to page out to virtual memory or have poor cache hit
ratios. However, if the problem size is too small, there is a limit to the divisibility of
the problem over the parallel nodes [2],[4].

- **Distributing problem input**
  Before an application can begin the parallel calculations, the input data must be
distributed to the processing nodes. The speed and efficiency in which the data sets
can be divided and communicated to the nodes is critical to the initialization time
of the application. If the communication channels among nodes is too slow, it may
be more efficient to keep the data and calculations local on a single processor than
to take the time to distribute the data for parallel processing [2],[15].

- **Algorithm serial percentage**
  As Amdahl’s law and Gustafson’s law indicate, the serial to parallel ratio of the
actual application has a significant impact on the scalability and efficiency of the
application in a parallel environment [7],[15],[21],[12],[11].

- **Parallel algorithm used**
  The algorithms utilized in the application affect the overall performance in a parallel
environment. For instance, a divide and conquer algorithm is well suited for parallel
processing. In terms of speedup and scalability analysis, incorrect uniformity between
the single-processor and multi-processor implementations can lead to an apparent
super-linear speedup caused by an algorithmic improvement [15],[12].

- **Operating system overhead**
  The design and decisions made by the operating system regarding the scheduling of
processes on the processor may influence the efficiency of a parallel environment.
Systems that have poor scheduling or force the application to frequently page fault or inefficiently switch contexts can experience performance degradation [4],[16],[10]. In addition, inefficiencies in system libraries and drivers may also impact overall performance [19].

- **Language and compiler characteristics**
  
  Due to the complexity of modern machine instruction sets, it is impractical to write an application in a low-level language for large scale computing. As such, the overall performance of any application is dependent on the ability of the language compiler or interpreter to efficiently translate the high-level programming language into machine code. While there are some programming practices to optimizing the resulting code, many of these methods, such as loop unrolling, are no longer as relevant due to the complexity of processor pipe-lining and branch prediction [1]. However, optimizing the code through the high-level language and an intelligent hardware specific compiler can ultimately increase the performance of a given application [4].

- **Application load time and memory initialization**

  Every application has an amortized cost of initialization. In parallel environments, especially with distributed memory, the initialization and allocation of memory may yield a speedup as the number of nodes increases because each node needs to allocate less memory (assuming Amdahl’s constant problem size approach). The load time of the application into executable memory, however, may increase due to code distribution among the processing nodes [2],[7].

2.3. Modern Scalability Analysis

In order to account for factors inhibiting efficiency, Shi suggests that an experimentally determined execution time approach be used to analyze parallel speedup. Alan Karp and Horace Flatt [11] formalize this approach. They determine speedup ($s$) with the following
formula based on execution time [15],[11],[12]:

\[ s = \frac{T(1)}{T(P)} \]  

where \( T(1) \) is the execution time using one processor, and \( T(P) \) is the execution time using \( P \) processors. Karp and Horace then define efficiency as:

\[ e = \frac{s}{P} \]  

The efficiency of a parallel system therefore indicates the percentage of theoretically obtainable speedup achieved. Since the speedup and efficiency are thus determined experimentally, the results show the perceived serial fraction of the parallel application. Instead of using the application instruction code to determine the parallel and serial portions as in Amdahl’s law and Gustafson’s law, the following formula can be used to discover the experimentally determined serial fraction \( f \) of the application [11]:

\[ f = \frac{\frac{1}{s} - \frac{1}{P}}{1 - \frac{1}{P}} \]

These formulations, however, assume that the problem size remains constant as more processing nodes are added. In most clusters, the memory of the parallel system increases as number of processing nodes increase, so as Gustafson [7], Sun, and Ni [18] point out, the problem size can also grow. To compensate for the increase in problem size or complexity as the number of processors increase, scaled speedup \( s_k \), where \( k \) is the problem size, is calculated as [11]:

\[ s_k = \frac{k \times T(1,1)}{T(P,k)} \]

where \( T(1,1) \) is the execution time of a single-processor node at the corresponding problem size, and \( T(P,k) \) is the execution time of the application using \( P \) processor nodes with
problem size \( k (k \geq 1) \). Scaled efficiency is then simply [11]:

\[
e_k = \frac{s_k}{P}
\]

(10)

The scaled serial fraction can then be calculated with [11]:

\[
f_k = \frac{\frac{1}{s_k} - \frac{1}{P}}{1 - \frac{1}{P}}
\]

(11)

Therefore, by executing an application on varying number of processing nodes, the efficiency of the parallel system can be calculated. While traditional benchmarks determine raw throughput, the proposed benchmark judges performance based on system efficiency. This allows the user to better identify barriers to performance and observe how effectively the parallel processing environment can be leveraged [11]. As such, the proposed benchmark application is not intended to replace other benchmarks such as LinPack, but it is meant to offer a better understanding of parallel systems through other metrics.

2.4. Other Benchmarks

2.4.1. LinPack

LinPack, and more importantly for parallel systems HPL (high performance LinPack), offers a standard metric for determining system performance. It uses an optimized linear algebra package called BLAS (basic linear algebra subprograms) to perform floating point calculations with a set algorithmic complexity. LinPack then returns a long series of performance estimates using varying data sets. All results are returned in GFLOP/s, or billion floating point operations per second. The final results are then reported in terms of [4]:

- \( R_{\text{max}} \) - GFLOP/s for the largest problem size executed
- \( N_{\text{max}} \) - problem size for \( R_{\text{max}} \)
- \( N_{\frac{1}{2}} \) - problem size at \( \frac{1}{2} \times R_{\text{max}} \)
- \( R_{\text{peak}} \) - the theoretically determined peak performance in GFLOP/s calculated by the formula (where \( \lambda \) is the number of operations per clock cycle, \( f \) is the processor
frequency, and $P$ is the number of processors):

$$R_{peak} = \lambda \times f \times P$$

These numbers are utilized to determine the top 500 high performance systems in existence. Submissions by candidates using the LinPack benchmark are ranked on the top500.org website semi-annually. Although these results show the overall throughput of floating point calculations on the system, they fail to identify system efficiency or the factors that may inhibit performance. As discussed in the LinPack paper, no single metric can truly reveal a system’s overall performance [4].

Another limitation to the LinPack benchmark is the inflexibility it has toward certain aspects of the application. LinPack uses 64-bit floating point calculations. Although many scientific applications may use such precision, a 16-bit or 32-bit architecture show slower performance than a 64-bit architecture. Many high performance applications are optimized for the available architecture, especially in regard to system word size. Consistently using a 64-bit calculation to compare maximum GFLOP/s throughput of systems may be biased towards certain architectures [4].

Despite LinPack’s immutable use of 64-bit calculations, the sizes and dimensional layouts of the data set used for the linear algebra calculations are user defined. It may take several attempts to fine tune the data size and layout in order to achieve a maximum GFLOP/s value from LinPack. This fine tuning is prone to human error and inconsistency, and as performance results vary drastically with the data sizes and layouts, a single or hurried execution of the LinPack benchmark yields unreliable results.

2.4.2. HINT

HINT stands for hierarchical integration, which is the algorithmic procedure it performs to benchmark a machine. The HINT benchmark improves on older benchmark designs and
offers new techniques and approaches to system analysis. HINT takes the function [9]:

\[
f(x) = \frac{(1-x)}{(1+x)}
\]

and integrates this function on the interval \(0 \leq x \leq 1\) using a hierarchical refinement method. The accuracy of this method improves over time. The HINT benchmark therefore measures performance not based on calculation throughput but rather on the accuracy or quality of the integration result per time interval. This measurement is called quality improvements per second, or QUIPS. Gustafson and Snell show that the QUIPS measurement is an accurate predictor for real application performance comparisons and other benchmark result comparisons. The QUIPS measurement also allows the HINT execution to be stopped at any point and still provide reliable results. The hierarchical integration should never reach a final answer, but always improve over time. This also gives HINT the ability to scale to any system or cluster size [9].

HINT incorporates several features that improve the accuracy of benchmarking and comparing systems. As Gustafson and Snell point out, modern systems have floating point arithmetic units that are far superior to their predecessors. The usage of pipe-lining, multiple cache levels, threading, specialized registers, and branch prediction have altered the performance factors of systems. Modern systems are now more limited by data management then by calculations. As such, the old benchmark ratings of GFLOP/s (billion floating point operations per second) and MIP/s (million operations per second) do not translate to actual system performance on real applications. The HINT algorithm and evaluation yields a metric that is universally comparable among machines. The HINT design also allows for the alteration of the primary data type used, such that the system can use integer or various floating point widths. As system word sizes, mantissa lengths, and processor abilities are so diverse, this feature allows HINT to better show the power of an overall system outside the influence of mathematical or compiler tricks integrated into the libraries or instruction sets. The HINT design also mimics real applications better than simple kernel benchmarks, using
an approximately 1 : 1 memory to operations ratio and performing a hierarchical approach that scientific applications often take [9].

HINT can produce a large amount of data since it shows the QUIPS per time interval as the benchmark progresses. Gustafson and Snell designated a formula, however, to reduce the HINT results to a single publishable number, called Net QUIPS. The Net QUIPS metric is the integral of the QUIPS curve over time obtained from the benchmark execution. This single metric can then be used to easily compare systems, similar to the GFLOP/s metric of LinPack, but with better accuracy [9].

2.4.3. NetPIPE

NetPIPE is a different type of benchmark, in that instead of providing a single system throughput metric, it shows the performance of a network with varying data transmission sizes. The design separates the benchmarking driver from the communication protocol to yield comparable results across different networking protocols. NetPIPE carefully tests data communication sizes ranging from a single byte up to a data size where the transmission takes a full second. Each test of a particular data transmission size is repeated several times and then averaged to ensure accurate time readings. This methodology allows for detailed and accurate network throughput graphs which can be used to identify network optimizing parameters, bottlenecks, and evaluate overall network performance. By separating the benchmark driver from the network protocol, and by using scaling size tests with no fixed maximum size, NetPIPE is scalable and portable to any environment [17]. Although it is strictly for the benchmarking of networks, the concepts presented and utilized by NetPIPE are ideal for a system level parallel processing scalability analysis benchmark.

2.5. Benchmarking Scalability and Efficiency

Most benchmarks attempt to provide metrics usable in the comparison of computational systems but are narrowly focused to particular system attributes such as overall calculation throughput or network communication. This thesis therefore proposes a simplistic, yet scalable algorithm to test parallel system efficiency. The application should use scalar data
widths directly associated with the system word width (e.g. 32-bit or 64-bit architectures). The algorithm should also yield accurate results without fine tuning of the data size or layout. The application should choose or suggest an ideal data set size and layout according to the specifications of the system. After choosing ideal parameters for the system, the benchmark should then methodically test parameter variations to obtain a more complete analysis of the parallel system. Comparing the variations to the base execution would then help diagnose limitations and bottlenecks to performance. The base execution should have the following optimizations:

- Single-precision calculations
- Atomic data width the same size as system word size (e.g. 32-bit or 64-bit)
- Processor nodes arranged to fit the data layout
- Data layout arranged to minimize processor cache misses
- Data size small enough to fit inside physical memory and avoid paging

This base case should be executed on an increasing number of nodes in the parallel processing system for system scalability analysis. Execution timings should be taken using $P$ processors where $N$ is the number of total nodes in the system $\forall P \in \{1, 2, 4, \ldots, \frac{N}{8}, \frac{N}{4}, \frac{N}{2}, N\}$. Doubling the number of processors for each execution provides sufficient results to plot the system’s scalability without having to test every value of $P$ from 1 \ldots N. Once the base execution results are obtained, comparisons can then be made for the following:

- Integer versus floating point calculations
- Atomic data width of $\omega$ versus $2\omega$ (where $\omega$ is the system word size)
- Linear versus random node ordering
- $(x \times y)$ versus $(2x \times \frac{y}{2})$ data layout

Comparing the execution times of these variations to the base execution case reveals information as to the effectiveness and efficiency of different system aspects. For instance, a drastic performance degradation seen when the processor ordering is randomized indicates
that the communication channel or network between the processors may have a particular geometric arrangement better suited for related geometrically arranged data.

A cellular automata (CA) algorithm was chosen for this proposed benchmark to meet these needs. A cellular automaton is a collection of interrelated cell units arranged in a lattice. Each unit’s current state transition is determined by the previous state of a collective stencil of neighboring units. Thus the entire cellular automaton evolves through a series of discrete states determined by the individual discrete unit transitions. The lattice structure of a cellular automaton may consist of any number of dimensions, with any discrete unit shape. Traditionally, most CA configurations use between one and three dimensions and are composed of regular polygons [13]. For the purposes of this benchmark a two-dimensional array cellular automata is used. The rule set determining state transitions averages the values of neighboring cells, thus attempting to re-balance the numerical weight across the array. Using this approach, the variations can be made in calculation complexity (integer vs. floating point), data size and shape, and the distribution of the data across the nodes. The next chapter discusses the implementation and program design of the proposed benchmark.
CHAPTER 3

BENCHMARK DESIGN

The benchmark proposed in this paper has two main goals. First, it should offer clear scalability analysis for the parallel system. This is achieved through the timing of executions using increasing fractions of the available processing nodes to determine speedup, efficiency, and experimental serial fraction. Second, it should identify potential performance bottlenecks in the system. This is accomplished through specific variations of the core algorithm and comparing the resulting execution times.

Since the motivation behind the development of this benchmark is to analyze the scalability of clusters, the benchmark has been dubbed CLUE, as it is a cluster evaluator. Unlike the LinPack or HINT benchmarks, the objective of CLUE is not to compare the calculation throughput of different systems, but rather to test and analyze attributes of a given parallel system. Benchmarks like HINT and LinPack should be used if a single performance metric is desired, or NetPIPE if the networking aspects of a system need testing. CLUE, however, is designed to show what parts of the system affect performance, rather than just how fast the system performs. CLUE can then be used in conjunction with HINT or LinPack to optimize the system attributes and thus obtain better QUIPS (quality improvements per second) or GFLOP/s (billion floating point operations per second) ratings.

3.1. Using Cellular Automata as a Benchmark

The CLUE benchmark is based on a cellular automata algorithm. According to Gustafson, benchmarks should be mathematically sound and operate algorithmically similarly to real applications [9]. The nature of cellular automata means that every iteration of the algorithm is predictable and verifiable. In addition, cellular automata algorithms are used for many simulation applications such as fluid dynamics, epidemiology, population growth, and forest fire analysis [13]. Modern systems are limited in performance due to data movement and
management rather than raw calculation speed [9]. Since cellular automata simulations are 
often memory intensive, they are well suited for benchmark applications.

The basic premise of a cellular automata algorithm is to transition a grid of cells from 
one state to the next on a cell by cell basis. The transition of the grid cells is determined 
by a rule set that is repeatedly applied to every cell. Rule sets typically incorporate a set of 
adjacent cells (known as a neighborhood or stencil) to the cell currently being evaluated [13]. 
A generic cellular automaton rule set, as in formula (13) with time t, cell \( C_{ij} \), and neighbors 
\( N_{ij} \), uses the states of the current cell and its neighbors at the previous time slice to derive 
the current state for the given cell.

\[
(13) \quad C_{ij}(t) \leftarrow (\forall f \in N_{ij}(t-1), C_{ij}(t-1))
\]

There are several parameters that can be adjusted based on this design to test various 
aspects of a system. For a given rule set, adjustments can be made to the grid height and 
width, neighborhood size, and cell data type or precision. Since general cellular automata 
algorithms are adjustable and are used for real simulations and scientific work, it is an ideal 
paradigm to leverage for benchmarking purposes.

To explain more thoroughly the inner workings of the cellular automaton used, consider 
the 16x16 grid shown in Figure 3.1. Although there are numerous options for grid layouts, 
neighborhoods, and rule sets, the focus here is on the options chosen for the CLUE bench-
mark's cellular automata algorithm. As stated earlier, the grid transitions that occur are 
composed of the individual cell transitions in the grid. The cell transitions are based on a rule 
set which incorporates a neighborhood of surrounding cells. In Figure 3.1, cell \( C_{ij} \) has eight 
neighbors each labeled \( N_{ij} \). Similarly, cell \( C_{p,q} \) has eight neighbors labeled \( N_{p,q} \) and \( C_{a,b} \) has 
neighbors \( N_{a,b} \). Note the dispersion of \( C_{a,b} \)'s neighbors. The CLUE benchmark uses a torus 
shaped grid, as shown in Figure 3.1. In a torus grid layout, each edge is considered adjacent 
to the edge on the opposite grid side. Thus, all eight cells labeled \( N_{a,b} \) are adjacent neighbors 
to \( C_{a,b} \).
The stencil arrangement shown in Figure 3.1 using eight adjacent neighbors is known as a Moore neighborhood. The rule set used in the CLUE benchmark takes the average of all neighbors to determine the next state value of a given cell. In other words, the transitional rule set for $C_{i,j}$ is:

\[(14) \quad C_{i,j}(t) \leftarrow \frac{\sum_{(a,b) \in N_{i,j}}(N_{a,b}(t-1))}{8}\]

Consider the algorithm to be a fluid dispersion simulation where each cell has a certain fluid pressure that over time is evenly dispersed through the grid. The algorithm thus needs to traverse the entire grid cell by cell to determine each cell’s next state value. The application is memory intensive both in overall memory size requirements and in the number
of memory reads required. This cellular automata paradigm tests a system’s data management capabilities. The computational intensity of the rule set can then test the system’s raw computational ability.

The rule set used in the CLUE benchmark, where a cell obtains the average value of its neighboring cells, allows for flexibility in that the data type used for storing each cell value, and the resulting calculation of the average can be based on integers, floating point values, or even double floating point values. Thus, not only can the data values be optimized for a system’s register width, but the difference between integer and floating point arithmetic can also be tested. This is similar to the HINT benchmark’s capability to choose the data type used on the calculations [9], but the CLUE benchmark uses this and other variation capabilities to enable comparisons of system components.

In order for scalability analysis to be valid, the algorithm being tested must remain constant irrespective of the number of processors used [8],[11],[12]. Cellular automata algorithms naturally have this property. If the grid in Figure 3.1 were to be divided among four processing nodes for speedup comparison to one node, the division could occur as depicted in Figure 3.2. Note that Figure 3.2 is exactly the same grid, but each node only has 3 of the total number of cells. This division allows the grid state transitions to occur a maximum of four times faster. As demonstrated by Amdahl’s law and Gustafson’s law [15],[7], this theoretic speedup is not obtainable. Even on a symmetric multi-processing architecture with four processors sharing the same memory, processor coordination and caching issues would degrade performance. In a cluster or other MIMD (multiple instruction, multiple data) architecture, communication latency among the processing nodes leads to a performance degradation when processing is delayed while data is shared among nodes. This communication is necessary to preserve the accuracy of the cellular automaton transitions. In Figure 3.2, all of cell $C_{p,q}$’s neighbors are located on node 3. However, cells near the node boundaries, like $C_{a,b}$ and $C_{i,j}$ require information from neighboring nodes in order to calculate the cell transitions.

Due to the adjacency of neighbors in the Moore neighborhood used here, each node need only communicate with other nodes which are directly adjacent. No cell in node 1 needs
information from node 3. Each node requires only the values contained in the adjacent edge of their neighbors. Node 1 only utilizes the information on the row labeled $B_3$ from node 2, and likewise node 2 requires only the row labeled $B_2$ from node 1. To expedite this sharing of values and optimize network buffering, the CLUE benchmark cellular automata algorithm has each node trade the entire edge row with the corresponding neighboring nodes at the start of each grid transition iteration. Each node then stores its neighbors’ edge rows in a local memory cache to avoid redundant value requests from other nodes. This methodology has been found to increase performance in parallel environments [16],[4].

For demonstration purposes, the grid in Figure 3.1 was split as shown in Figure 3.2 keeping the same overall 16x16 size. This methodology would follow the speedup model in accordance with Amdahl’s law. The CLUE benchmark, however, increases the overall data size linearly with the increase in processing nodes, such that in the example of Figure 3.1, instead of splitting the grid in fourths, each of the nodes would obtain an entire 16x16 grid. This follows the speedup model described by Gustafson, which is model used for calculations throughout the CLUE benchmark [7].

3.2. Designing a Good Benchmark

The premise for CLUE is straightforward: utilize a simple algorithm to test system scalability. Then, alter aspects of the application and compare the results of the altered version. The difference in performance due to the alteration then helps identify how certain system components affect overall performance. Figure 3.3 diagrams the application alterations and comparisons that can be made.

The CLUE benchmark consists of two applications: a C++ cellular automata timing test, and a Perl script. The user executes the Perl script which analyzes the system to determine ideal parameters, then begins to execute the C++ timing tests with different parameters. The timing tests return comma separated lists of values for the executions. Although the values obtained directly from the C++ program are fully usable for manual analysis, the Perl script takes these values and performs a comparative analysis between the timing tests and creates
a comprehensive result set in XML (extensible markup language) format. The XML can then be viewed directly in raw form, imported into other analysis applications, or rendered with an XSLT (extensible style sheet language transformations) document to provide immediate graphical and tabular results for user analysis.

In order for a benchmark to be usable, it must be able to scale and be portable across many systems. The core of the benchmark is the C++ timing application, which is easily portable to any system and architecture. C++ is also optimized for the particular system via

![Figure 3.2. A cellular automata grid spread across 4 nodes.](image)
the compiler. The libraries used in the C++ application depend solely on C++ and POSIX (portable operating system interface) standards, and the parallel environment used such as MPI (message passing interface) or PVM (parallel virtual machine).

The Perl component of the CLUE benchmark is designed for the user to obtain understandable benchmark results with little effort. It is customizable, and the default system values can be overridden via command line parameters and a keyword/value-based configuration file. For systems that do not support Perl, however, the user can opt to execute the C++ component with various parameters manually to obtain basic results. The comma separated values that are returned from the C++ application can then be manually analyzed via a spreadsheet or statistical application.

Figure 3.4 shows the layout of the Perl component of the CLUE benchmark. Upon execution, the Perl script first processes the configuration file and command line parameters.
1: Process Config File
2: Process Command Line Parameters
3: Gather System statistics
4: Determine Optimal Settings
5: for $M \leftarrow 1, 2, 3, \ldots$, Trial Repetitions do
6: for all $V \in$ (Program Variations) do
7: for $N \leftarrow 1, 2, 4, 8, \ldots$, Number of Total Nodes do
8: TempResults $\leftarrow$ exec("mpirun(nodes $\leftarrow N)$, clue.exe(Variation $\leftarrow V$)"
9: if (TempResults faster than Results($V$)) then
10: Results($V, N$) $\leftarrow$ TempResults
11: end if
12: end for
13: end for
14: end for
15: ScalabilityAnalysis(Results($\forall N \in \{1, 2, 4, 8, \ldots$, Number of Total Nodes\}$))
16: ComparativeAnalysis(Results($\forall V \in$ Variations))
17: PrintCSV(Results)
18: PrintXML(Results)

Figure 3.4. CLUE Perl wrapper algorithm.

and then queries the system to determine ideal execution parameters for the C++ timing tests. Note that the Perl wrapper contains three loops. The loop starting on line 7 tests the particular variation on different number of nodes in the cluster. By testing each variation across different node counts, the effects of the variation changes can be better analyzed. The the loop beginning on line 6 provides the variations to be tested. The loop on line 5 repeats the entire series of tests multiple times to ensure accuracy. In lines 9-10, the program retains the information for the fastest execution for each variation and node count test combination. This helps ensure that only optimal timings are used for calculations and eliminates issues caused by periodic or anomalous system activities during the benchmark. Each of these tests executions need to be executed non-consecutively so that a performance limiting system event does not compromise all tests repetitions.

Once all timing tests are complete, the CLUE Perl application performs analysis calculations on the data sets and prints the results in both comma separated value and XML forms.
The comma separated value format can then be imported into statistical packages for analysis. The XML report can be rendered with an XSLT style sheet for easy reviewing of the benchmark’s results.

Note that the CLUE Perl application is neither computationally nor memory intensive. It merely executes the optimized C++ cellular automata application which is the core of the CLUE benchmark. The cellular automata algorithm used in the CLUE timing tests is designed to allow maximum flexibility for comparisons of system components. Although algorithmic improvements can be made to increase the transition speed of most cellular automata algorithms [1], the CLUE benchmark follows a straightforward algorithm that loops over the entire grid every iteration. Figure 3.5 shows the pseudo-code for the CLUE C++ timing test program.

The C++ application is designed in three main parts: the main cellular automata algorithm, a matrix template class, and a process timing and statistics class. The process timing class contains three primary functions: Timer.Start(), Timer.Stop(), and PrintTimes(). The Timer.Start() function in line 3 takes a snapshot of the process information, such as current time, system time, user time, number of page faults, and number of context switches. The Timer.Stop() function in line 15 then takes another snapshot of the process information and subtracts the values obtained from the Timer.Start() function call. The PrintTimes() function in line 16 then prints out the results. In the parallel context of MPI or PVM [6],[5], the class first gathers the data collected from each process of the parallel application and returns the average, standard deviation, minimum, and maximum values for each of the metrics. The CLUE Perl script can then collect these metrics for analysis and reporting.

The matrix template class is designed to allow any data type to be used by the cellular automata algorithm. This class also incorporates the necessary structures and values needed for a parallel environment like MPI. The class contains a function called Trade Node Edges() wherein each node trades its grid edges with adjacent neighbors, as called on line 5 of Figure 3.5. To adapt the benchmark for other parallel environments like PVM or Mosix, the matrix class contains the majority of the environment specific functions.
1: `InitializeMPI()`
2: `GRID < Type > ← Generate Random Grid(XSize,YSize)`
3: `Timer.Start()`
4: for \(i \leftarrow 1, 2, \ldots, \text{Iterations}\) do
5: \(\text{Trade Node Edges}()\)
6: for \(x \leftarrow 1, 2, \ldots, \text{XSize}\) do
7: \(\text{for } y \leftarrow 1, 2, \ldots, \text{YSize} \) do
8: \(\text{for all } n \in \text{Neighborhood} \) do
9: \(\text{SUM} \leftarrow \text{SUM} + \text{GRID} < \text{Type} > (x_n, y_n)\)
10: end for
11: \(\text{GRID}(x, y) \leftarrow \text{TypeCast}\left(\frac{\text{TypeCast(SUM)}}{\text{TypeCast(Number of Neighbors)}}\right)\)
12: end for
13: end for
14: `Timer.Stop()`
15: `PrintTimes()`

Figure 3.5. CLUE core benchmarking algorithm.

The CLUE Perl application passes the grid width, height, number of iterations to perform, and other parameters to the C++ benchmark application. After processing parameters and initializing the parallel environment, the application instantiates a grid on each node with random values. As the algorithm has been tested and proved correct, and storage I/O is not a target of this benchmark, loading an existing grid and storing the results is impractical for the sake of performance testing. In addition, the use of randomly generated grid starting values is useful since the grid sizes and data types can vary with each execution.

Once the environment and the data set has been initialized in line 2, the timing and statistics class takes the beginning snapshot of the process metrics. The application then proceeds through a set number of grid state transitions as seen in line 4 of Figure 3.5. At the beginning of each iteration, the nodes exchange edge rows and cache their neighbors’ values locally. Each node then iterates across the grid calculating each cell transition. The calculations that occur on line 11 of insure that the proper data type is used. This allows the application to stay within the data type specified to the cell values, using integer calculations or floating point calculations accordingly.
After completing the specified number of grid state transitions, the application stops the process timers (line 15), calculates and gathers the resulting metrics, and prints the results to standard output (line 16). This output is then captured by the CLUE Perl script for later analysis and reporting. In the absence of the CLUE Perl script, the results can be captured and analyzed by other processes.

3.3. Scalability Analysis

Scalability analysis shows the efficiency and speedup of a parallel system as more processors are added to the application. CLUE reports values for analyzing the scalability of the parallel system in both a high-level summary of all the application variations and for each variation individually. The summary information showing the speedup, efficiency, and serial fraction using the base line application variation shows how the parallel system performs overall as various fractions of the available processing nodes are utilized. These results can help to determine if adding more nodes to the system is cost effective. If the efficiency is poor, the system may be better utilized in smaller groups of processing nodes rather than in its entirety. Review of the values returned for the individual application variation metrics and comparisons between the variations can help to determine what system components lead to the most performance loss as the application scales.

3.4. Systematic Testing and Comparative Analysis

While scalability analysis is useful in determining appropriate usage and expansion levels for a parallel system, it is also important to understand where performance is lost in a system. Identifying bottlenecks to performance can help to elevate those barriers and increase the overall performance level of the system. In order to identify the impact of different system components, CLUE systematically tests variations of the timing algorithm for comparison.

In order to ensure accurate results, each variation is executed several times. The exact number of variation executions is determined by the a parameter in the configuration file. After each execution of a particular variation, the execution times are compared and only the
fastest time is retained. This assures that anomalous system activity has less impact on the final results of the benchmark.

As discussed previously and shown in Figure 3.3, the parameters of the cellular automata algorithm and data set are varied by the Perl script, and the results are gathered and presented in XML form. By examining the results of the CLUE benchmark, the effects of various system components to the overall throughput and efficiency can be determined. In order to make these comparisons though, a base line must first be established. When the Perl script for CLUE is executed, it first queries and examines the system to determine the ideal conditions for the cellular automata algorithm. These ideal factors include:

- Physical memory size - the amount of RAM (random access memory) attached to a single processing node
- Processor cache size - the amount of cache per processor (e.g. L1 and L2 data cache in x86 architecture)
- System word size - the main register size for the processor (e.g. 32-bit or 64-bit)
- Number of nodes - the number of processing nodes in a cluster, or processors in SMP (symmetric multi-processing) systems

While the Perl script attempts to determine these factors automatically, the values can be overridden using command line parameters or the configuration file. With these values, the Perl script can customize the base line execution for optimal performance. For instance, the Perl component of CLUE attempts to execute the base line timing test such that:

- The data width of each grid cell is equal to the width of the system word (or register) size in order to optimize throughput per clock cycle.
- The entire memory footprint per node (grid, tracking variables, etc) fits within the node’s physical memory with a small amount to spare in order to avoid virtual memory swapping.
• The width of the grid is less than \(\frac{1}{3}\) the processor cache size in order to minimize cache misses by keeping each cell in local cache from the first time it is needed, to the last time per iteration.

• The size of the local neighborhood fits within the first level of processor cache to minimize L1 cache misses by keeping each cell in local cache from the first time to the last time it is read per row pass.

• The nodes are geometrically arranged in order of node numbering, especially for non-switched networks (e.g. node 2 is between nodes 1 and 3).

• The data type of each grid cell uses floating point calculations. Note that although this may not be optimal for all systems, it is used for the base line since most scientific applications utilize floating point arithmetic.

By comparing the variations to the ideal case, the performance impact of the various system components can be identified. After executing all the variations several times to ensure accurate results, CLUE calculates additional derived metrics and prints the results for user analysis.

3.5. Reporting

Upon completion of all the trials, the CLUE Perl wrapper contains execution times, system metrics, and parameter information for each execution. The Perl wrapper then uses the speedup, efficiency, and serial fraction formulas discussed in Section 1 to create the respective derived metrics for each execution. In addition, the average cell state transitions per second (ACT/s) are calculated for each execution using the formula:

\[
ACT/s = \frac{Grid\ Height \times Grid\ Width \times iterations}{wall\ time}
\]  

(15)

The ACT/s metric shows the number of cell state transitions per second averaged over all nodes. Thus, the ACT/s number is already normalized for any cluster size. Total system throughput can be derived by multiplying the ACT/s measurement of formula (15) by the
number of nodes $N$ used:

\begin{equation}
Net \ ACT/s = N \times ACT/s
\end{equation}

Averages are then taken for each node count, each variation type, and overall system averages for the system. The results are then reported both in a comma separated file, and an XML file. The averages for the node counts and variation groupings are useful for analyzing the individual system. The system level averages can be used to evaluate and compare different systems. Although the ACT/s metric does not equate directly to metrics like GFLOP/s or MIP/s, it is useful in determining overall system throughput between systems. Comparing systems according to their ACT/s and efficiency metrics gives a better picture between different systems.
CHAPTER 4

RESULTS AND ANALYSIS

The CLUE benchmark was tested on two parallel systems – a sixteen node and an eight node cluster – which were varied using different network connections and code compilations. Comparing and analyzing these variations in the clusters creates a controlled test platform to ensure CLUE can properly reveal system efficiencies and the potential causes of inefficiencies.

4.1. Result Metrics

Table 4.1 shows the configurations of the tested clusters. The Compiler Opt row lists slow and fast, meaning compiled with no optimization using the -i386 architecture target flag, and compiled with optimization and the -pentium4 architecture target flag respectively. All cluster variations used 32-bit Intel® Pentium®4 processors1 with Linux®2 kernel version 2.6.13. The C++ programs on clusters A, C, and F were compiled with GCC (GNU®3 Compiler Collection) version 4.0.2, and the programs on clusters B, D, E, and G were compiled with GCC version 3.3.5. All clusters used LAM/MPI 7.1.1 environments.

The CLUE benchmark was executed on each of these nodes using settings of twenty iterations (or whole grid transitions) and ten trial attempts. Only the fastest time of the ten attempts was recorded. The resulting metrics are shown in tables 4.2 and 4.3 which show the ACT/s (average cell state transitions per second), efficiency, and serial fraction by node counts and parameter variations.

For the application parameter variations, the base line variation consisted of the cellular automata cell averaging rule set using single precision floating point calculations where each node had an identically sized square grid. The cellular automata neighborhood used the Moore neighborhood, or the eight adjacent surrounding cells (as shown in Figure 3.1).

---

1Intel Corporation, www.intel.com
2Linus Torvalds, www.linuxmark.org
3Free Software Foundation, www.gnu.org
<table>
<thead>
<tr>
<th></th>
<th>Cluster A</th>
<th>Cluster B</th>
<th>Cluster C</th>
<th>Cluster D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>2.0 GHz</td>
<td>1.5 GHz</td>
<td>2.0 GHz</td>
<td>1.5 GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>512 KB</td>
<td>256 KB</td>
<td>512 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>RAM</td>
<td>1024 MB</td>
<td>512 MB</td>
<td>1024 MB</td>
<td>512 MB</td>
</tr>
<tr>
<td>Network Speed</td>
<td>100 Mb</td>
<td>1000 Mb</td>
<td>100 Mb</td>
<td>1000 Mb</td>
</tr>
<tr>
<td>Network Media</td>
<td>Switch</td>
<td>Fiber Switch</td>
<td>Switch</td>
<td>Fiber Switch</td>
</tr>
<tr>
<td>Compiler Opt</td>
<td>fast</td>
<td>fast</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>Grid Size</td>
<td>6697 x 6697</td>
<td>4716 x 4716</td>
<td>6697 x 6697</td>
<td>4716 x 4716</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Cluster E</th>
<th>Cluster F</th>
<th>Cluster G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>8</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>1.5 GHz</td>
<td>2.0 GHz</td>
<td>1.5 GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>256 KB</td>
<td>512 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>RAM</td>
<td>512 MB</td>
<td>1024 MB</td>
<td>512 MB</td>
</tr>
<tr>
<td>Network Speed</td>
<td>100 Mb</td>
<td>10 Mb</td>
<td>10 Mb</td>
</tr>
<tr>
<td>Network Media</td>
<td>Switch</td>
<td>Hub</td>
<td>Hub</td>
</tr>
<tr>
<td>Compiler Opt</td>
<td>fast</td>
<td>fast</td>
<td>fast</td>
</tr>
<tr>
<td>Grid Size</td>
<td>4716 x 4716</td>
<td>6697 x 6697</td>
<td>4716 x 4716</td>
</tr>
</tbody>
</table>

Table 4.1. Cluster tested configurations.

The integer variation was identical to the base line except each cell stored integer values instead of floating point. This kept the memory size at one system word per cell, but forced the application to use integer arithmetic when performing the averaging computation. Since many modern processors have a separate floating point processing unit or coprocessor, this variation allows comparison between the floating point and integer capabilities of a system’s processor.

The double precision variation uses double word sized floating point values for each cell’s storage, thus taxing the system’s floating point computational capability, bus, and processing registers. In addition to creating more expensive computations than the base line variation, the use of the double sized variable type tests how well a system can handle atomic data types larger than the system word size. Some systems may have extra wide buses, registers, and specialized instruction sets to handle such data types, but others may have to take multiple steps to separate the value into smaller parts, process them, and then recombine. Comparing the results of this variation with the base line identifies how well the system can handle
<table>
<thead>
<tr>
<th>Cluster</th>
<th>ACT/s (Million)</th>
<th>Efficiency</th>
<th>Serial Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cluster A</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12.184</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>12.053</td>
<td>98.92%</td>
<td>1.09%</td>
</tr>
<tr>
<td>4</td>
<td>12.023</td>
<td>98.68%</td>
<td>0.45%</td>
</tr>
<tr>
<td>8</td>
<td>11.978</td>
<td>98.31%</td>
<td>0.25%</td>
</tr>
<tr>
<td>16</td>
<td>11.990</td>
<td>98.41%</td>
<td>0.11%</td>
</tr>
<tr>
<td><strong>Cluster B</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4.765</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>4.757</td>
<td>99.83%</td>
<td>0.17%</td>
</tr>
<tr>
<td>4</td>
<td>4.759</td>
<td>99.87%</td>
<td>0.04%</td>
</tr>
<tr>
<td>8</td>
<td>4.725</td>
<td>99.16%</td>
<td>0.12%</td>
</tr>
<tr>
<td><strong>Cluster C</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.138</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>5.127</td>
<td>99.80%</td>
<td>0.20%</td>
</tr>
<tr>
<td>4</td>
<td>5.129</td>
<td>99.83%</td>
<td>0.06%</td>
</tr>
<tr>
<td>8</td>
<td>5.123</td>
<td>99.71%</td>
<td>0.04%</td>
</tr>
<tr>
<td>16</td>
<td>5.115</td>
<td>99.56%</td>
<td>0.03%</td>
</tr>
<tr>
<td><strong>Cluster D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3.962</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>3.950</td>
<td>99.71%</td>
<td>0.29%</td>
</tr>
<tr>
<td>4</td>
<td>3.931</td>
<td>99.21%</td>
<td>0.27%</td>
</tr>
<tr>
<td>8</td>
<td>3.924</td>
<td>99.04%</td>
<td>0.14%</td>
</tr>
<tr>
<td><strong>Cluster E</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4.770</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>4.745</td>
<td>99.48%</td>
<td>0.52%</td>
</tr>
<tr>
<td>4</td>
<td>4.729</td>
<td>99.14%</td>
<td>0.29%</td>
</tr>
<tr>
<td>8</td>
<td>4.738</td>
<td>99.33%</td>
<td>0.10%</td>
</tr>
<tr>
<td><strong>Cluster F</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12.181</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>11.767</td>
<td>96.60%</td>
<td>3.52%</td>
</tr>
<tr>
<td>4</td>
<td>11.608</td>
<td>95.30%</td>
<td>1.64%</td>
</tr>
<tr>
<td>8</td>
<td>11.463</td>
<td>94.10%</td>
<td>0.90%</td>
</tr>
<tr>
<td>16</td>
<td>11.156</td>
<td>91.58%</td>
<td>0.61%</td>
</tr>
<tr>
<td><strong>Cluster G</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4.763</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>2</td>
<td>4.695</td>
<td>98.57%</td>
<td>1.45%</td>
</tr>
<tr>
<td>4</td>
<td>4.635</td>
<td>97.31%</td>
<td>0.92%</td>
</tr>
<tr>
<td>8</td>
<td>4.585</td>
<td>96.25%</td>
<td>0.56%</td>
</tr>
</tbody>
</table>

Table 4.2. CLUE results by processor count for the base line variation.
<table>
<thead>
<tr>
<th>Cluster</th>
<th>ACT/s (Million)</th>
<th>Efficiency</th>
<th>Serial Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Line</td>
<td>11.978</td>
<td>98.31%</td>
<td>0.25%</td>
</tr>
<tr>
<td>Data Rearrange</td>
<td>11.956</td>
<td>98.14%</td>
<td>0.27%</td>
</tr>
<tr>
<td>Double Precision</td>
<td>11.672</td>
<td>98.60%</td>
<td>0.20%</td>
</tr>
<tr>
<td>Integer</td>
<td>16.873</td>
<td>99.02%</td>
<td>0.14%</td>
</tr>
<tr>
<td>Cluster B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Line</td>
<td>4.725</td>
<td>99.16%</td>
<td>0.12%</td>
</tr>
<tr>
<td>Data Rearrange</td>
<td>4.725</td>
<td>99.21%</td>
<td>0.11%</td>
</tr>
<tr>
<td>Double Precision</td>
<td>4.681</td>
<td>98.72%</td>
<td>0.19%</td>
</tr>
<tr>
<td>Integer</td>
<td>6.028</td>
<td>99.33%</td>
<td>0.10%</td>
</tr>
<tr>
<td>Cluster C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Line</td>
<td>5.123</td>
<td>99.71%</td>
<td>0.04%</td>
</tr>
<tr>
<td>Data Rearrange</td>
<td>5.097</td>
<td>98.98%</td>
<td>0.15%</td>
</tr>
<tr>
<td>Double Precision</td>
<td>4.308</td>
<td>92.87%</td>
<td>1.10%</td>
</tr>
<tr>
<td>Integer</td>
<td>6.667</td>
<td>99.40%</td>
<td>0.09%</td>
</tr>
<tr>
<td>Cluster D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Line</td>
<td>3.924</td>
<td>99.04%</td>
<td>0.14%</td>
</tr>
<tr>
<td>Data Rearrange</td>
<td>3.901</td>
<td>98.48%</td>
<td>0.22%</td>
</tr>
<tr>
<td>Double Precision</td>
<td>1.963</td>
<td>98.85%</td>
<td>0.17%</td>
</tr>
<tr>
<td>Integer</td>
<td>4.574</td>
<td>98.74%</td>
<td>0.18%</td>
</tr>
<tr>
<td>Cluster E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Line</td>
<td>4.738</td>
<td>99.33%</td>
<td>0.10%</td>
</tr>
<tr>
<td>Data Rearrange</td>
<td>4.704</td>
<td>98.81%</td>
<td>0.17%</td>
</tr>
<tr>
<td>Double Precision</td>
<td>4.683</td>
<td>98.44%</td>
<td>0.23%</td>
</tr>
<tr>
<td>Integer</td>
<td>5.997</td>
<td>98.91%</td>
<td>0.16%</td>
</tr>
<tr>
<td>Cluster F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Line</td>
<td>11.463</td>
<td>94.10%</td>
<td>0.90%</td>
</tr>
<tr>
<td>Data Rearrange</td>
<td>10.755</td>
<td>88.54%</td>
<td>1.85%</td>
</tr>
<tr>
<td>Double Precision</td>
<td>10.593</td>
<td>88.65%</td>
<td>1.83%</td>
</tr>
<tr>
<td>Integer</td>
<td>15.705</td>
<td>92.62%</td>
<td>1.14%</td>
</tr>
<tr>
<td>Cluster G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Line</td>
<td>4.585</td>
<td>96.25%</td>
<td>0.56%</td>
</tr>
<tr>
<td>Data Rearrange</td>
<td>4.428</td>
<td>92.92%</td>
<td>1.09%</td>
</tr>
<tr>
<td>Double Precision</td>
<td>4.404</td>
<td>92.66%</td>
<td>1.13%</td>
</tr>
<tr>
<td>Integer</td>
<td>5.783</td>
<td>95.30%</td>
<td>0.70%</td>
</tr>
</tbody>
</table>

double precision computations and taxes the network interconnecting the processing nodes with larger data transmission sizes.
The data rearrangement variation takes the $X \times Y$ grid on each node and elongates it to $(2X) \times \frac{Y}{2}$. The new grid has exactly the same physical memory size and is targeted to still optimize processor cache. The elongated grid has the same memory width as the double precision. This doubles the size of the shared borders between nodes, increasing transmission size across the network. Comparing this to both the base line and double precision variations identifies the cost of the network to system efficiency, and clarifies the impact of the double precision’s variability in both transmission size and calculation complexity.

All of the clusters used to test CLUE for this work used the same centralized network interconnection topologies. Thus, the node rearrangement variation proved ineffective for these tests. The design and intent of this variation is to test the impact of node arrangements in alternative networking topologies. Figure 4.1 shows several different networking topologies. The base line CLUE variation optimizes the node arrangement by forcing nodes to trade only with their numerically adjacent neighbors. In a switch topology, rearranging the node numbering would have little impact on performance. However, forcing the cluster nodes to communicate with neighbors which are not physically adjacent in a network such as the ring or grid networks would cause extra network latency as messages must be relayed among nodes to reach the proper destination node.

4.2. Performance Analysis

The graph shown in Figure 4.2 illustrates the ACT/s throughput values for these variations for each of the clusters. In every cluster, the executions using integer calculations outperformed those using floating point computations. Likewise, the double precision calculations had a slower throughput than the single precisions calculations. This verifies that the floating point calculations, and especially the double precision calculations, do cost more computationally. The data rearrangement seemed to only have a noticeable impact to the ACT/s throughput for clusters $F$ and $G$ which utilized a hub for the network interconnection. This shows that the extra network traffic incurred by the elongation of the grid is noticeable on less efficient networking architectures.
Examining Figure 4.3 and Figure 4.4 shows an inverse proportion between the efficiency and experimental serial fraction for each of the variations on all clusters. This should be expected as the efficiency and serial fraction are inversely related both in formula and logic.

Where in Figure 4.2 the clusters had a similar hierarchy of highest and lowest ACT/s numbers across the variations, the efficiency graphs of Figure 4.3 (and similarly the serial fractions in Figure 4.4) reveal uniqueness between the clusters. Analyzing the efficiency of the application variations shows the characteristics of the cluster performance more clearly than throughput (ACT/s) alone.

Figure 4.3 shows that cluster B is more efficient than cluster A, whereas in Figure 4.2, cluster A was more than twice as powerful as cluster B in terms of ACT/s. Although cluster B is not as powerful overall, it is using its resources more efficiently. In addition, the similarity between the Figure 4.3 graph shape for cluster B and cluster C (which used
a non-optimized compilation) along with the similar throughput of cluster B and cluster C in Figure 4.2 indicates that perhaps the operating system or libraries were not optimized or compiled specifically for the processor. Inefficiencies in system libraries or devices drivers are known to cause performance degradation [19].

Figure 4.3 also shows a significant drop in efficiency for the double precision calculations in cluster C. Clusters F and G not only show a similar inefficiency for double precision computations, but also for the data rearrangement variation. Since only double precision floating point calculation variation was severely affected for cluster C, it can be deduced that cluster C is impaired by more difficult floating point computations. As cluster C used a compilation of CLUE that was not optimized by the compiler and which the compiler targeted for an older architecture instead of the particular processor used by the cluster nodes, this assumption holds.
Figure 4.3. Efficiency by variation.

However, since both the double precision and data rearrangement variations were affected in cluster F and cluster G, the cause of inefficiency is more likely the network than the processing unit. Both of these variations have twice the data transmission requirements than the base line variation. This conclusion is valid since these two clusters used a 10 Mb hub for their network interconnection compared to the faster switched networks used on the other clusters.

4.3. Scalability Analysis

Instead of showing metrics for each application variation, Figure 4.5 and Figure 4.6 show the results when various number of processing nodes were utilized. Viewing the results along this dimension allows for scalability analysis of the clusters.

Note that in Figures 4.5 and 4.6 every cluster shows a decrease in efficiency as more nodes are used in the computation. This is expected and follows the approximate predictions of formula (1) and formula (2). Unlike in Figure 4.3 and Figure 4.4, these two graphs do
not show an inverse relationship between efficiency and serial fraction. Instead, the serial fraction decreases proportionally with the efficiency as more processors are added. This follows Gustafson’s assertion that as the problem size grows with the number of processors, the serial portion of the application is overshadowed by the increasingly larger parallel portion [7].

Table 4.2 shows that cluster C is more efficient than cluster A, even though cluster A has a higher ACT/s throughput. In addition, the serial fraction and efficiency decrease at a slower rate as more processors are leveraged on cluster C compared to cluster A. This indicates that cluster C is better suited to handle additional processors.

In comparison, cluster F does not scale well as more processors are added. Note that cluster F has a fast processor, but slow network, whereas cluster C and cluster D have slower processors (or in this case, code which is not optimized and targeting a lower powered processor), but a fast network. These results indicate that the network connecting processing
nodes has a strong impact on the overall efficiency of a parallel system. The similar efficiency and serial fraction results for cluster B and cluster E, however, indicates that increasing the network capabilities of a system does not always yield significant improvements in performance or efficiency. Bottlenecks in other system components, such as processing power, prevent the more powerful network from being fully utilized. Utilizing a 100 Mb switch on cluster E instead of the 10 Mb hub on cluster G showed a significant improvement in efficiency. However, leveraging the 1000 Mb fiber switch for cluster B failed to show a significant improvement over cluster E.

4.4. Summary of Results

Observing the results produced by CLUE on the seven test clusters verified that CLUE shows the efficiency and scalability of clusters. The various metrics produced are indicative of the unique characteristics of each cluster. Although the throughput of the clusters in terms of ACT/s was a useful metric, the efficiency and serial fraction metrics helped to better
understand the actual performance characteristics of each cluster. The variance shown in the CLUE metrics correlate strongly with the expected system performance for the tests on the given clusters. Using these correlations, CLUE can now be applied to other clusters where system inefficiencies and bottlenecks to performance are less obvious.
CHAPTER 5

SUMMARY AND CONCLUSION

5.1. Summary

The CLUE benchmark is designed to reveal a system’s performance not based on a single throughput metric, but by examining a parallel system’s efficiency in scalability and providing metrics which can help to determine the system components which inhibit performance. Through the examination of seven different cluster variations, it has been shown that the CLUE benchmark results do aid in identifying system components which may inhibit performance. This thesis proves the validity of the CLUE approach to system analysis and benchmarking. Now that CLUE has been shown to provide accurate information, it can be expanded to look at other system components, used in other environments, and automate more of the optimization and result analysis.

5.2. Future Work

While the current implementation of the CLUE benchmark proves the validity of the scaled and comparative analysis approach, it is still limited in scope. CLUE has only been proven on Linux systems using the MPI clustering environment. The modular design of the application allows for CLUE to be expand and used in more diverse environments. Additional alternative executions could reveal other system information yet undiscovered.

5.2.1. Expanding Environments

For the scope of this thesis, CLUE was only tested using one MPI message passing environment. The design, however, allows for any process coordination environment to be used. Modules for parallel environments such as PVM and OpenMosix could thus be developed and implemented into CLUE. CLUE could then be used to evaluate various parallel environments and library implementations.
5.2.2. Expanding Tests

CLUE reveals information pertaining to processor scalability and the performance impact of:

- Integer calculations
- Floating point calculations
- Network interconnections
- Node arrangements

While these are crucial to parallel system performance, other variations could be developed to better test other system components. For instance, adjusting the cellular automata neighborhood to a four neighbor stencil, or using eight random non-adjacent neighbors instead of the current neighborhood of eight adjacent cells could reveal the impact of the system's tiered memory scheme. As the current variations all attempt to optimize local processor cache hits, forcing the program to retrieve data from main memory more frequently would reveal the performance impact of processor cache misses.

The CLUE benchmark variations currently all use buffered network communication. Forcing each node to communicate one cell at a time instead of the entire border row at once would show the impact of the networking stack and related buffers [14]. While a benchmark like NetPIPE would reveal more detailed information about the underlying network, incorporating tests in CLUE for the network stack would provide more information comparable to the entire parallel system.

5.2.3. Expanding Analysis

CLUE currently reports system metrics and leaves the interpretations to the user. Although the reporting of these raw metrics are crucial for proper system analysis, functionality could be added to the Perl wrapper to extrapolate observations for a system. Comparing the variance in efficiency, serial fraction, and throughput metrics, basic conclusions could be automatically drawn for the different application parameters used. For instance, a decrease in throughput occurring in the double-precision but not the data rearrangement variation would
indicate poor floating point performance, especially for larger word sizes. Likewise, a slight
decrease in throughput accompanied by a significant reduction in efficiency as more nodes
are utilized would be indicative of poor networking or inter-process communication. While
human analysis is still imperative for proper system evaluation, some automated conclusions
may help to expedite analysis and to give more understandable results to non-technical users.

5.2.4. Further Testing

As CLUE is expanded, gathering results from different cluster types would be useful for
ensuring the accuracy of the results and optimizing the overall application. Currently, CLUE
has only been tested on Linux clusters using an MPI clustering environment on switched
ethernet networks. Results from other clusterable operating systems would help to prove the
validity of the base algorithm and implementations across different systems. Results from
clusters with a grid, torus, hypercube, or ring network topology would help to determine
the node rearrangement variation validity. Perhaps most importantly, tests on clusters with
larger node counts would help to determine system scalability accuracy. CLUE has only been
tested on small to moderate cluster sizes, and although the results on these smaller clusters is
sufficient to draw valid conclusions, clusters with node counts at or above 1024 nodes should
also be examined.

5.2.5. User Interface

When CLUE is executed on a system, it displays real-time information including the current
elapsed time, estimated time remaining, and percent completed for the entire benchmark.
The benchmark execution must finish, however, before usable results can be viewed. The
ability to send a control signal to the CLUE Perl wrapper should be added, which forces the
application to finish the test currently being processed and then proceed with the partial result
aggregation without completing all the tests.

In addition, CLUE currently uses system information to determine the ideal grid size, but
not the number of iterations nor the number of individual trial attempts. The iterations and
trial attempts used must be defined in the configuration file or by command line parameters.
Performing a quick timing test on a single node could give the Perl wrapper sufficient information to determine the ideal execution length for each test. Observing the accuracy and difference between individual timed trial attempts could allow the Perl wrapper to determine when a adequate number of tests have executed for accurate results. Automating the parameters based on system information would ease the user’s configuration efforts and potentially generate more precise results. However, the user should still have the ability to specify other parameter values to the benchmark via the configuration file and command line arguments.

5.3. Conclusion

Performing scalability analysis aids in the evaluation of system efficiency and can contribute to the decisions concerning a parallel system’s computational infrastructure investment. Evaluating the performance of various aspects of a system also contributes to a better understanding of performance hindrances and which components of a system can or should be optimized. CLUE provides this information through automated testing of the parallel system. Unlike other benchmarks that simply report overall system throughput, CLUE also reveals important information about the efficiency of the system and the various components that comprise the system. Future versions of the CLUE benchmark may be used in numerous parallel environments and show even more system component information. CLUE is a new benchmark which demonstrates a better comparison among different parallel systems than existing benchmarks and can diagnose where a particular parallel system can be optimized.
BIBLIOGRAPHY


