Benchmarking the CLI for I/O-Intensive Computing *

Xiao Qin Tao Xie
Department of Computer Science
New Mexico Institute of Mining and Technology
Socorro, New Mexico 87801
http://www.cs.nmt.edu/∼xqin

Ahalya Nathan Vijaya K. Tadepalli
Department of Computer Science
and Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska 68588-0115

Abstract

Common Language Infrastructure, or CLI, is a standardized virtual machine, which increasingly becomes popular on a wide range of platforms. In this paper we developed three I/O-intensive benchmarks for the CLI using various techniques. The first benchmark is designed in accordance with an application behavioural model that rebuilds the behavior of real world I/O-intensive applications. The second benchmark is a trace driven simulator that simulates five I/O-intensive applications. The third benchmark is a micro I/O-Intensive benchmark used to emulate a simple web server. In addition, the performances of the benchmarks are evaluated on the SSCLI. The results suggest that the CLI is a potential virtual machine for I/O-intensive computing.

1. Introduction

The Common Language Infrastructure, or CLI, enables applications written in multiple high level languages to be executed in different system environments without rewriting the applications to factor in unique characteristics of the environments. The Common Language Infrastructure provides a virtual execution environment similar to the one developed by Sun Microsystems for Java programs. The CLI uses a compiler to process language statements into an intermediate form of executable code called bytecode. When a program is running, its bytecode is compiled on the fly into the native code recognized by the machine architecture of a given computer.

* This paper appeared in the Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05), the 6th Int'l Workshop on Parallel and Distributed Scientific and Engineering Computing, IEEE/ACM, April 4-8, 2005.

The main goal of the CLI is to provide an easy way for designers and programmers to write components and applications from any language. This goal can be achieved by carefully defining a standard set of types, attempting to make various components self-describing. Therefore the CLI is able to provide a high performance common execution environment by allowing compilers to fully utilize the execution environment. An execution engine provides the execution environment for executing managed code. In addition, the execution engine manages components, isolation model, and several run-time services. Due to aforementioned advantages, the CLI increasingly becomes popular on a diversity of platforms.

In what follows, we describe four main areas in the CLI specifications:

1. Common type system: To support the complete implementation of a variety of programming languages, the common type system provides types and operations found in many programming languages.
2. Common language specification: the common language specification, an agreement between language designers and class-library designers, specifies a subset of the common type system and a set of usage conventions. With the common language specification in place, it becomes efficient for users to access the class library.
3. Virtual execution system: The virtual execution system enforces the common type system by loading and running programs written for the CLI.
4. Metadata: Metadata, which is independent of any particular programming language, is used to describe and reference types defined by the common type system.

In the last decade input and output data sizes of parallel applications have continued growing at a

fast pace, and a vast majority of applications have become I/O-intensive in nature. Typical examples of I/O-intensive applications include long running simulations [9], archives of raw and processed remote sensing data, and out-of-core applications [1], to name just a few. These applications share a common feature in that their disk I/O requirements are extremely high. For example, out-of-core applications have high demands to access data sets that exceed the capacity of physical memory, and it has become conventional wisdom to develop out-of-core applications in a way to explicitly handle data movement in and out of core memory avoiding the use of virtual memory [2]. To design and implement high performance I/O-intensive applications to meet the needs of both scientific and non-scientific computing, benchmarking I/O-intensive applications on a diversity of computing platforms plays an important role.

Due to the popularity of CLI and high demands of I/O-intensive applications, the goal of our study is to investigate the possibility of leveraging the CLI virtual machines to support I/O-intensive computing. To achieve this goal, we developed and evaluated three benchmarks that are I/O-intensive in nature. Building the benchmarks helps designers and programmers of I/O-intensive applications to gain a greater understanding on the CLI.

The results of our work show that the CLI is an efficient virtual machine for I/O-intensive computing, indicating that I/O-intensive applications written in a wide range of programming languages like C# can be executed on multiple platforms without modifying source code. In this paper we focus on designing and developing benchmarks for I/O-intensive workloads in the context of the CLI. We only developed and tested three benchmarks using SSCLI [8] on Windows XP platform. Executing the benchmarks on other platforms such as Mac OS X 10.2 is out of the scope of this research.

We took three independent approaches in an attempt to devise I/O-intensive benchmarks for the CLI. The first benchmark is designed based on an application behavioral model that rebuilds the behavior of real world I/O-intensive applications. The second benchmark is a trace driven simulator that simulates five I/O-intensive applications. The third benchmark is a micro I/O-Intensive benchmark used to emulate an I/O-intensive web server. In addition, the performances of the benchmarks are quantitatively evaluated on the SSCLI.

The rest of the paper is organized as follows. In Section 2, an application behavioral model is given. Based on the model, the simulation of a real world I/O-intensive application is presented. We evaluated the performance of the application on the SSCLI. Section 3 presents a second benchmark simulating both scientific and non-scientific applications. Section 4 proposes a third micro benchmark, which is a simple web server. Finally, Section 5 summarizes the main contributions of this paper and comments on future directions for this work.

2. Simulation of an I/O-intensive Application

2.1. Application Behavioral Model

To simulate real I/O-intensive applications, we introduce an application behavioral model derived from a parallel program model proposed by Rosti et al. [7]. The parallel program model captures features and interactions of CPU and I/O activities in a representative class of parallel scientific applications [13]. We extended their model by considering communication requirements imposed by parallel applications [14][15]. Our model is important and desirable in the sense that the first benchmark design makes full use of this model to quickly emulate a parallel application running on the CLI. In addition, our model is reasonably general because it is applicable for both I/O- and communication-intensive parallel applications. Before describing the model, we present the following terms defined in [7].

1. Program: a parallel application is composed of a set of programs,
2. Phase: each disjoint interval composed of an I/O burst followed by a computation burst and possibly followed by a communication burst within a program,
3. Working set: a sequence of consecutive phases that are statistically identical.

The execution behavior of a program is therefore comprised of a sequence of working sets and an application consists of a set of interdependent programs that execute in a coordinated manner [7]. Note that programs of an application may exhibit different I/O and communication behaviors. Similarly, phases of a program may impose different I/O and communication requirements [7]. Let N be the number of phases for a program (task) of a parallel application running on a node, and T^i be the execution time of the ith phase. T^i can be obtained by the following equation, where T^i_{CPU}, T^i_{COM}, and T^i_{Disk} are the time spent on CPU, communication, and disk I/O in the ith phase:

$$T^i = T^i_{CPU} + T^i_{COM} + T^i_{Disk} \quad (1)$$

Thus, the total execution time of the program is quantitatively estimated by the following equation:

$$T = \sum_{i=1}^{N} T^i \tag{2}$$

Let R_{CPU}, R_{COM}, and R_{Disk} be the requirements of CPU, communication, and disk I/O, and the requirements can be measured by the following three equations:

$$R_{CPU} = \sum_{i=1}^{N} T^i_{CPU}, \tag{3}$$

$$R_{Disk} = \sum_{i=1}^{N} T^i_{Disk}, \tag{4}$$

$$R_{COM} = \sum_{i=1}^{N} T^i_{COM}. \tag{5}$$

We are now in a position to consider the execution time model for a parallel job (the terms job, and application are used interchangeably). Given a program, its behavior information can be formally represented by a vector with M working sets:

$$\vec{\Gamma} = [\Gamma_1, \Gamma_2, ..., \Gamma_M] \tag{6}$$

The ith working set of $\vec{\Gamma}$ is a vector with four parameters:

$$\Gamma_i = (\phi_i, \gamma_i, \rho_i, \tau_i) \tag{7}$$

where ϕ_i denotes I/O fraction, γ_i represents communication fraction, ρ_i is relative execution time, and τ_i indicates the number of phases in the ith working set. I/O fraction is defined as the fraction of the nth phase that represents the length of the phase's I/O burst. Similarly, communication fraction is defined as the fraction of the nth phase that represents the length of the phase's communication burst. Relative execution time of the nth phase is the ratio between the nth phase execution time and the total execution time of the program.

An example of the above notation is illustrated in Figure 1, where the program has four working sets. Figure 1a show the phase behavior of a program with respect to absolute time and Figure 1b show the phase behavior of the program with respect to relative execution time. The initial single-phase working set is composed of a large fraction of time in reading data from a disk (i.e., $\Gamma_1 = (\phi_1 = 0.52, \gamma_1 = 0.29, \rho_1 = 0.287, \tau_1 = 1)$). The program execution then has a working set consisting of two phases that are communication intensive (i.e., $\Gamma_2 = \Gamma_3 = (\phi_2 = 0, \gamma_2 = 0.85, \rho_2 = 0.185, \tau_2 = 2)$), followed by another single-phase working set that both computationally intensive and communication intensive (i.e., $\Gamma_4 = (\phi_4 = 0, \gamma_4 = 0.57, \rho_4 = 0.194, \tau_4 = 1)$). The program ends with a single-phase working set during which the results are written to disk (i.e., $\Gamma_5 = (\phi_5 = 0.81, \gamma_5 = 0, \rho_5 = 0.148, \tau_5 = 1)$).

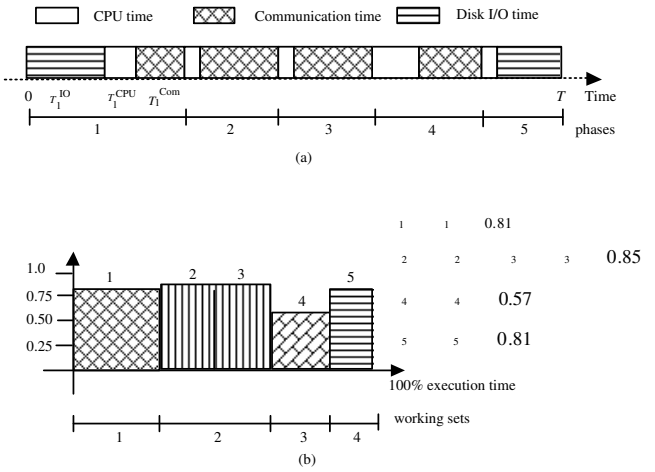

Figure 1. Example of program behavior with N=5 phases: (a) absolute execution time (b) relative execution time $\vec{\Gamma}$ =[((0.52, 0.29, 0.287, 1), (0, 0.85, 0.185, 2), (0, 0.57, 0.194, 1), (0.81, 0, 0.148, 1)].

2.2. The I/O-intensive Application

Based on the model presented above, we simulated an I/O-intensive application referred to as QCRD [7][12]. The QCRD application is developed by C language, attempting to solve the Schrodinger equation for the cross sections of the scattering of an atom by a diatomic molecule. The application is I/O-intensive in nature, because it needs to store large size of the global matrices into memory and the data has to be processed iteratively. A second reason for burst I/O behavior is because the application first fills a set of buffers in memory and then process the data stored in the buffers. Note that the I/O activities in the QCRD application follow a cyclic pattern, and the application consists of two independent programs denoted by $\vec{\Gamma}_1$ and $\vec{\Gamma}_2$.

Using the model presented in Section 2.1, we are able to formally describe the QCRD application as below:

$$\vec{\Gamma} = [\vec{\Gamma}_1, \vec{\Gamma}_2,] \tag{8}$$

For the first program, the application has a sequence of CPU- and I/O-intensive phases, which repeat 12 times [7]. Thus, we have:

$$\vec{\Gamma}_1 = [\Gamma_{1,1}, \Gamma_{1,2}, ..., \Gamma_{1,24}] \quad (9)$$

where $\Gamma_{1,i} = (0.14, 0, 0.066, 1)$ for $i = 1, 3, ...23$; and $\Gamma_{1,i} = (0.97, 0, 0.0082, 1)$ for $i = 2, 4, ..., 24$.

In the second program, there are 13 identical phases with more I/O intensive activities [7]. Formally, the program is characterized as:

$$\vec{\Gamma}_2 = [(0.92, 0, 0.03, 13)] \quad (10)$$

2.3. Experimental Results

We simulated the QCRD application on the SSCLI, and this section presents the experimental results. The goal of this experiment is to analyze the performance of the simulated QCRD application serving as a benchmark on the SSCLI. Due to the space limits, we only study the performance of QCRD, a representative I/O-intensive application. The development of other simulated applications is subject to our future work.

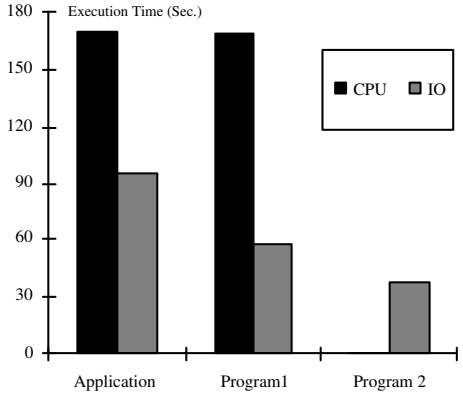

Figure 2. Execution time of computation and disk I/O for the QCRD application, and two programs.

Figure 2 plots the execution times of computation and disk I/O for the QCRD application as well as its two independent programs. Figure 3 depicts the percentage of execution time for both computation and disk I/O processing. Figures 2 and 3 indicates that the I/O activities in the second program is more intensive compared with that in the first program. Figure 3 shows that the QCRD application spends a noticeably large amount of time on I/O processing. It is

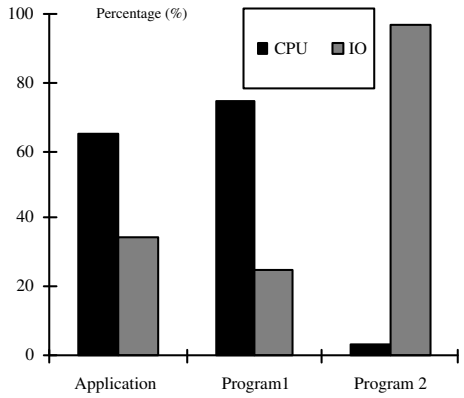

Figure 3. Percentage of execution time for computation and disk I/O.

to be noted that compare the simulated result with that generated from a real implementation, the error rate is less than 10%. We attribute the simulation error to the system instabilities and non-dedicated environment.

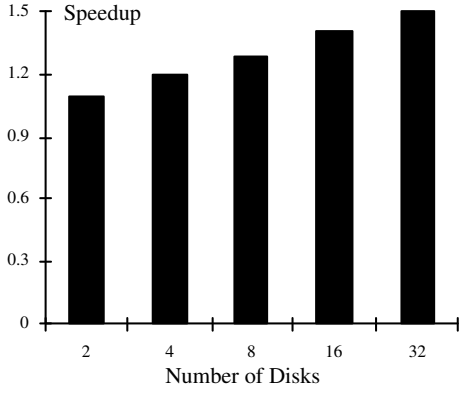

Figure 4. Speedup of the application as a function of the number of disks.

We turn our attention to the efficiency of the QCRD application. Figure 4 plots the speedup of the application as a function of the number of disks in the simulated system. We observe from Figure 4 that the speedup changes slightly with the increasing value of the disk number. This result implies that increasing the number of disks does not necessarily improve the performance of the QCRD application. The reason is because the speedup is dominated by the first program of the application, and the first program runs longer

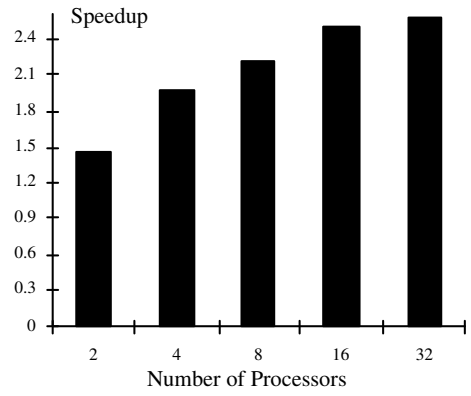

Figure 5. Speedup of the application as a function of the number of CPUs.

than the second program. Since the first program is more CPU-intensive than I/O-intensive, it is expected to efficiently improve the performance of QCRD by increasing the number of CPUs in the system. This argument can be confirmed by Figure 5 that illustrates the speedup of the application as a function of the number of CPUs. Note that the results are consistent with those reported in the literature. This study shows that application developers can leverage the model presented in Section 2.1 to evaluate the performance of I/O- and communication-intensive applications without spending a huge amount of time implementing the applications.

3. A Benchmark using I/O Traces

3.1. I/O trace files

While in Section 2 we have successfully simulated an I/O-intensive application using a model, in this section we intend to present simulation results for a number of I/O-intensive applications. We have two sets of trace files, scientific and non-scientific, collected from the University of Maryland [10]. We used these trace files to analyze the behavior of the system in the SSCLI when I/O (read, write, open, close, seek) operations are issued to a large file containing 1GB of data.

Trace files corresponding to both scientific and non-scientific applications are used in our experiments. The domains of scientific applications include remote-sensing, linear algebra, electron scattering, rendering planetary pictures, quantum chemistry, and radar imaging. The non-scientific applications include a relational database, data-mining, a parallel web server and textual search. More precisely, we developed a benchmark on the SSCLI to simulate the following I/O-intensive applications:

1. Data mining (Dmine): This application extracts association rules from retail data [6].
2. Parallel text search (Pgrep): This application is used for partial match and approximate searches. It is a modified parallel version of the agrep program from the University of Arizona [11].
3. LU decomposition (LU): This application computes the dense LU decomposition of an out-of-core matrix [5].
4. Titan: This is a parallel scientific database for remote-sensing data [3].
5. Sparse Cholesky (Cholesky): This application is capable of computing Cholesky decomposition for sparse, symmetric positive-definite matrices [4].

3.2. Structure of the Trace File

The trace file header contains parameters for number of processes, number of files, number of records, offset to the Trace records and the sample file on which the I/O operations will be issued. Each trace record contains parameters corresponding to the I/O operation to be performed (Open =0,Close=1,Read=2,Write=3,Seek=4), number of records for which the I/O operation need to be performed, process id, field, wall clock time, process clock time, offset, length.

3.3. Experiment setup

Our simulator reads each trace files (scientific, non-scientific) and performs the I/O operations on a local disk. Both read and write operations have an offset provided such that I/O operations could be performed from that offset. Seek operations are performed from the beginning of the file to the offset as mentioned in the trace files. Timing is taken for opening, closing, reading, writing, seeking in a file to analyze the behavior of I/O operations.

3.4. Experimental Results

The simulation results are summarized in Tables 1-4 in this section.

1. Datamining trace file consists of synchronous I/O reads with the average read time and seek time recorded.
2. Titan trace file consists of synchronous I/O reads with the average read time recorded.

Appl. name	Data size (Bytes)	Read time (ms)	Open time (ms)
Data Mining	131072	0.0025	0.0006
	Close time (ms)	**Seek time (ms)**	
	0.0072	7.88E-05	

Table 1. Results for the data mining application

Appl. name	Data size (Bytes)	Read time (ms)	Open time (ms)
Titan	187681	0.002	0.0005
	Close time (ms)		
	0.005		

Table 2. Results for the titan application

3. LU Factorization trace file consists of synchronous I/O reads with the seek and write time recorded. The open and close time for the LU application are 0.0006 and 0.4566 ms, respectively.

Request number	Data size (Bytes)	Seek Time (ms)
1	66617088	9.43E-05
2	66092544	7.54E-05
3	64518912	9.69E-05
4	63994368	7.27E-05
5	62945280	0.0002
6	60322560	9.60E-05

Table 3. Results for the LU application

4. Sparse Cholesky Factorization has a number of synchronous I/O operations. The open and close time for this application are 0.00067 ms and 0.0071 ms, respectively.

The execution times were measured for opening, closing, reading, writing, and seeking operations. We observed from the experiments that for all trace files the time spent closing a file was longer than the time taken to open the file. When the file is opened, a page or two is placed in I/O buffers.

At the time when a read, write, or seek operations is performed, a prefetch operation will be invoked accordingly. In case where the respective region is not present in the buffers, the corresponding pages are fetched from the disk to the memory. Therefore, I/O operations in light of prefetching experience relatively high execution times. For example time taken to seek 62945280 bytes

Request number	Data size (Bytes)	Seek time (ms)	Read Time (ms)
1	4		7.33E-05
2	28044		7.54E-05
3	28048		0.0169
4	133692		7.27E-05
5	136108		0.01
6	143452		0.01
7	132128		0.025
8	149052		0.015
9	144642		0.004
10	84140	7.92E-05	
11	217832	8.26E-05	
12	624548	8.16E-05	
13	916884	7.92E-05	
14	1592356	8.15E-05	
15	2018308	0.00012	
16	2446612	7.54E-05	

Table 4. Results for the Cholesky application

from the LU Factorization file takes is longer when compared against the times spent reading 63994368 or 64518912 bytes. The same observation is made for the Sparse Cholesky Factorization File, in which reading 28048 bytes takes more time than reading 133692 bytes. This is because a page fault occurs, resulting in the corresponding page being fetched from the disk into the buffers.

4. Micro Benchmark: A Multi-threaded Web Server

This section is focused on a micro benchmark that generates a large number of I/O operations. Specifically, the micro benchmark simulates a multi-threaded web server that intensively issues read and write operations to a local disk. The multi-threaded web server tested on the SSCLI is useful for performing a comprehensive file system analysis on a variety of computing platforms.

4.1. Design of the micro benchmark

A main thread of the web server initializes the system by creating a separate thread to handle each client connection. The main thread continues accepting new connections. For each request received by the server, the incoming data is read into a buffer and parsed for request type and file name. If the request type is "GET", then the required file is read and sent back to the client. When the request is "POST", the data delivered from the client is written to a file. Each request received by the server results in reading a file or writ-

ing to an existing file or creating a new file. In addition, the number of threads increases with the increasing number of clients.

The server starts listening on port 5050 using TcpListener class. Once a new request is received, it accepts the connection by using AcceptSocket(), which returns a socket descriptor. This socket is passed to another class called "work", which receives the socket. A new thread is created and StartListen() method in the "work" class is invoked when creating the new thread. The thread is started by calling Start() function. In StartListen() function, a network streams is created for the socket, and then incoming data is received into a byte array which is converted into a string later. This string is parsed for "GET" or "POST" depending on GET or POST, either doGET() or doPost is invoked. In doGet() method, the requested file is read and sent to the client through the socket. In the case of doPost(), the data is written to a new file created by using a random number generator. Hence, no synchronization is required for write operations. The data is stored to the new file using streamwriter class. Times spent in performing the read and write operations are measured using QueryPerformanceCounter, which gives time elapsed in milliseconds. The time taken for performing the read operation includes: (1) creating an instance of filestream class, (2) reading the data from the file, and (3) closing the filestream.

4.2. Experimental Results

A number of image files are used for the purpose of conducting experiments. The sizes of each file are 50607 bytes, 7501 bytes, and 14063 bytes. The experimental results are shown in the following table.

Request number	Data size (Bytes)	Read Time (ms)	Write Time (ms)
1	7501	2.1175	2.8538
2	50607	2.2319	2.7442
3	14603	1.6764	2.4026

Table 5. Response time of read and write operations

The results plainly show that the first file I/O operation by the server takes more time than the subsequent read or write operations. Table 6 shows details of read operation performed multiple times on the same file.

Figure 6 plots the response time of read operations as a function of data size. The results show that the

Trail number	Data size (Bytes)	Read Time (ms)
1	14063	9.0181
2	14063	6.7331
3	14603	6.5070
4	14063	7.4598
5	15063	5.9489
6	14603	3.2441

Table 6. Response time of read and write operations

time spent in reading a file for the first time is greater than that taken for subsequent read operations. This observation can be explained by the following reasons:

1. I/O buffers are utilized to store data loaded by the first read operation. Consequently, subsequent read operations on the same data are allowed to read data from the buffers rather than the local disk, thereby shortening the response time.
2. There is a delay caused by the JIT compiler when the web server is handling the first read or write request. This might force the program to start the disk I/O operations relatively late for the first time, because functions are compiled only when they are required.
3. Files loaded from the local disk system can be partitioned into multiple pages. Hence, there is a likelihood that subsequent files are placed in one of the pages that has been prefetched into the I/O buffers. In other words, no swapping of pages occurs and, therefore, time taken to perform file I/O operation is reduced.

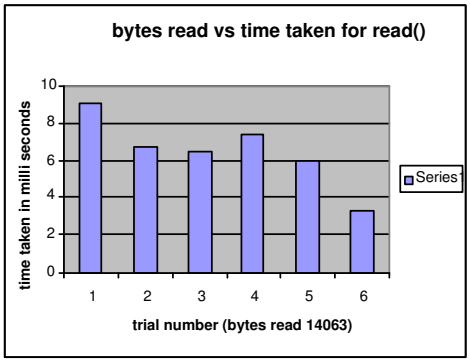

Figure 6. Data size (measured in Bytes) vs. response time of read operations.

5. Conclusion

The common language infrastructure is a standardized virtual machine, which are widely used on various platforms. Due to the popularity of the CLI and high demands of I/O-intensive applications, it is important and fundamental to study the possibility of leveraging the CLI virtual machines to support I/O-intensive computing. To quantitatively evaluate performance of I/O-intensive applications running on the CLI, we developed three benchmarks for the CLI using a variety of techniques. In particular, the first benchmark is designed based on an application behavioral model, the second benchmark is a trace-driven simulator for a collection of real world I/O-intensive applications, and the third benchmark emulates a simple web server. This study is intended to provide designers and programmers of I/O-intensive applications a fundamental understanding on the CLI. The experimental results indicate that the CLI allows I/O-intensive applications written in a wide range of programming languages to be executed in a diversity of computing environments without rewriting the applications.

Due to time and space limits, we only present the empirical results of the three benchmarks on the SSCLI. As a future research direction, we plan to evaluate performance of the benchmarks for I/O-intensive computing on other virtual machines like java virtual machine. Furthermore, we intend to develop benchmarks for I/O-intensive computing in a widely distributed environment, and compare the performance of the benchmarks on different CLI-based virtual machines.

6. Acknowledgements

This work was partially supported by a start-up research fund (103295) from the research and economic development office of the New Mexico Institute of Mining and Technology.

References

[1] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Trans. on Computer Systems*, 19(2):111–170, May 2001.

[2] M. M. Cettei, W. B. L. III, and R. B. Ross. Support for parallel out of core applications on beowulf workstations. In *Proceedings of the 4th IOPADS*, March 1998.

[3] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: A high-performance remote-sensing database. In *Proc. of International Conference on Data Engineering*, 1997.

[4] A. A. et al. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the 4th IOPADS*, pages 15–27, May 1996.

[5] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15(5), Sept 1994.

[6] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. In *Technical Report CS-TR-3515, University of Maryland, College Park*, August 1995.

[7] E. Rosti, G. Serazzi, E. Smirni, and M. s. Squillannte. Models of parallel appllications with large computation and I/O requirements. *IEEE Trans. Software Engineering*, 28(3):286–307, March 2002.

[8] D. Stutz, T. Neward, and G. Shilling. Shared source CLI. In *OReilly and Associate, Cambridge, MA*, 2003.

[9] T. Tanaka. Configurations of the solar wind flow and magnetic field around the planets with no magnetic field: Calculation by a new mhd. *Journal of Geophysical Research*, pages 17251291–17262, Oct 1993.

[10] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O systems for parallel machines: An application-driven study. In *Technical Report CS-TR-3802, University of Maryland, College Park*, May 1997.

[11] S. Wu and U. Manber. agrep - a fast approximate pattern-matching tool. In *USENIX Conference Proceedings*, pages 153–162. TeX Users Group, Winter 1992.

[12] Y.-S. Wu, S. Cucaro, P. Hipes, and A. Kuppermann. Quantum chemical reaction dynamics on a highly parallel supercomputer. *Theoretica Chimica Acta*, 79:225–239, 1991.

[13] L. T. Yang. The improved CGS method for large and sparse linear systems on bulk synchronous parallel architectures. In *Proceedings of the 5th International Conference on Algorithm and Architectures for Parallel Processing (ICA3PP-02)*, pages 232–237, Beijing, P.R. China, October 23-25, 2002.

[14] L. T. Yang and R. Shaw. The IBiCGStab method on bulk synchronous parallel architectures. In *Proceedings of the 16th International Symposium on High Performance Computing Systems and Applications (HPCS-02)*, pages 139–146, Moncton, New Brunswick, Canada, June 16-19, 2002.

[15] T. Yang and H. X. Lin. Performance analysis of the IQMR method on bulk synchronous parallel architectures. *Journal of Supercomputing*, 13(2):191–210, March/April 1999.