On the Calibration, Verification and Validation of an Agent-Based Model of the HPC Input/Output System

Diego Encinas, Marcelo Naiouf, Armando De Giusti

Informatics Research Institute LIDI
CIC's Associated Research Center
Universidad Nacional de La Plata
50 y 120, La Plata, 1900, Argentina
Email: {dencinas, mnaiouf, degiusti}@lidi.info.unlp.edu.ar

Sandra Mendez

Computer Sciences Department
Barcelona Supercomputing Center (BSC)
Barcelona, 08034, Spain
Email: sandra.mendez@bsc.es

Dolores Rexachs and Emilio Luque

Computer Architecture and Operating Systems Department
Universitat Autònoma de Barcelona
Bellaterra, 08193, Spain
Email: {dolores.rexachs, emilio.luque}@uab.es

***Abstract*—High Performance Computing (HPC) applications can spend a significant portion of their execution time making Input/Output (I/O) operations into files. Improving I/O performance becomes more important for the HPC community as parallel applications produce more data and use more compute resources. One of the methods used to evaluate and understand the I/O performance behavior of such applications in new I/O system or different configurations is using modeling and simulation techniques. In this paper, we present a simulation model of the HPC I/O system by using Agent-Based Modelling and Simulation (ABMS) based on the functionality of the I/O Software Stack. Our proposal is modeled using the concept of white box so that the specific behavior of each of the modules or layers in the system can be observed. The I/O software stack layers are analyzed using code instrumentation for the features corresponding to I/O operations and calibration of the initial model. The verification and validation of an initial implementation has shown a similar behavior between the measured and simulated values for the Interleaved or Random (IOR) benchmark by using different file sizes.**

Keywords–Agent-Based Modelling and Simulation (ABMS); HPC-I/O System; Parallel File System.

I. INTRODUCTION

Many scientific applications benefit considerably from the rapid advance of processor architectures used in the modern High Performance Computing (HPC) systems. However, they can spend a significant portion of their execution time making Input/Output (I/O) operations into files. Inefficient I/O is one of the main bottleneck for scientific applications in a large-scale HPC environment.

In the HPC field, the I/O strategy recommended is the parallel I/O that is a technique used to access data in one or more storage devices simultaneously from different application processes so as to maximize bandwidth and speed up operations. For its implementation, a parallel file system is required; otherwise the file system would probably process the I/O requests it receives sequentially, and no specific advantages in relation to parallel I/O would be gained.

Generally, evaluating the performance offered by a HPC I/O system with different configurations and the same application allows selecting the best settings. However, analyzing application performance can also be a useful before configuring the hardware.

One of the methods used to predict different application configurations behavior in a computer system is using modeling and simulation techniques. That is, analyzing and designing simulation models based on the parallel I/O architecture allows reducing complexity and fulfilling application requirements in HPC by identifying and evaluating the factors that affect performance.

There are several research efforts in HPC I/O system simulators focusing on storage architecture and some layers of the I/O software stack. The Simulator Framework for Computer Architectures and Storage Networks (SIMCAN)[1] is oriented to optimizing communications and I/O algorithms. The Parallel I/O Simulator of Hierarchical Data (PIOSimHD) [2] was developed to analyze Message Passing Interface-Input/Output (MPI-I/O) performance. The Co-design of Exascale Storage System (CODES) [3] is a framework developed to evaluate the design of the exascale storage systems. The High-Performance Simulator for Hybrid Parallel I/O and Storage System (HPIS3) [4] models application workload.

CODES and HPIS3 are based on Rensselaer's Optimistic Simulation System (ROSS) [5], which is a parallel simulation platform. SIMCAN was developed using OMNET++, and PIOSimHD was programmed in Java. All the tools mentioned use an event-based simulation paradigm (Discrete Event Simulation, DES). We propose to develop a simulator using Agent-Based Modeling and Simulation (ABMS) that will allow evaluating the performance of the I/O software stack. The agent paradigm is used in various scientific fields and is of special interest in Artificial Intelligence (AI), it allows successfully solving complex problems compared with other classic techniques [6]. It is a simulation technique that recreates the functionality of different components in a real system by modeling entities known as agents. Basically, an agent is an entity capable of perceiving and acting based on changes in its environment. It can also interact with other agents, executing and coordinating its actions, to achieve goals.

 ISBN: 978-1-61208-756-6

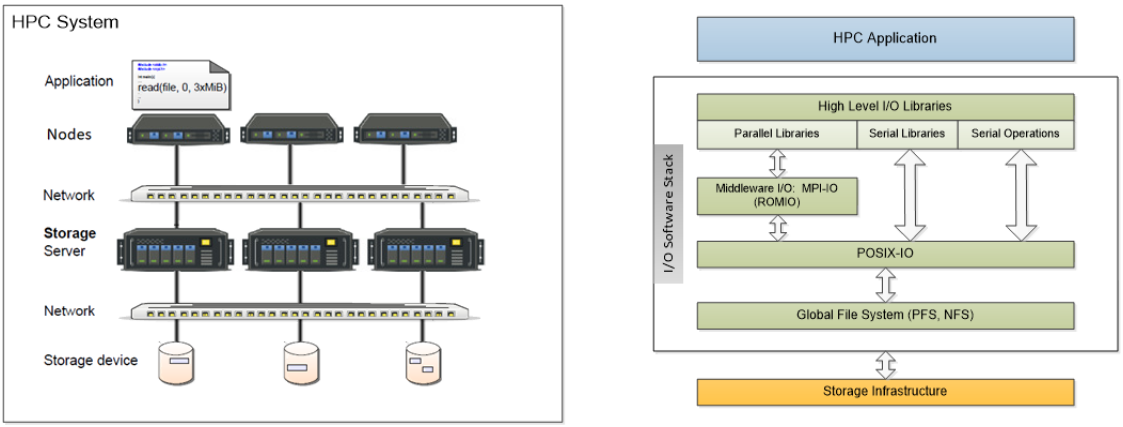


Figure 1: A typical HPC System and the I/O Software Stack

Generally, both paradigms operate in discrete time, but DES is used for low to medium abstraction levels. In ABMS, system behavior is defined at an individual level, and global emergent behavior appears when the communication and interaction activities among the agents in an environment start. In fact, ABMS is easier to modify, since model debugging is usually done locally rather than globally [7].

An advantage of ABMS is that different types of models could be created for each part of the system [8][9]. This is useful since the behaviors of the models differ from each other as they are related to diverse actions like processing, communications and storage. Furthermore, the environments could be both software and hardware. ABMS allows it to implement different components in a modular and flexible way, affording the possibility of connecting and disconnecting different parts of a complex system for a layer-level analysis.

In this paper, we present a model of the HPC-IO system for an initial simulation using ABMS. Our proposal is modeled using the concept of white box so that the specific behavior of each of the modules or layers in the system can be observed. The I/O software stack layers were analyzed using code instrumentation for the features corresponding to I/O operations and calibration of the initial model. The verification and validation of the proposed model has shown similar I/O behavior on the real and simulated environment.

The rest of this paper is organized as follows. Section II briefly describes I/O system in HPC. Section III addresses a functionality analysis for the development of the initial conceptual model. Section IV describes the proposed model, initial implementation and validation. Finally, Section V presents our conclusions and future works.

II. BACKGROUND

The I/O subsystem in the HPC area consists of two abstraction levels, software and hardware. Usually, the I/O Software includes parallel file system and high level I/O libraries and the I/O hardware refers to storage devices and networks. However modern HPC I/O system can include more components increasing the complexity of the I/O system.

Figure 1 illustrates the structure of the I/O software stack. An I/O operation goes through the software stack from the user application up until it obtains access to the disk from where data are read or on which data are written. Since this parallelism is complex to coordinate and optimize, the implementation of intermediate several layers was designed as a solution.

A. HPC I/O Strategies

The most common I/O strategies in HPC are the serial or parallel accesses into files. Serial I/O is carried out by a single process and it is a non-scalable method because operation time grows linearly with the volume of data and even more with the number of processes, since more time will be required to collect all data in a single process [10].

Parallel I/O usually presents two methods or variations of them: ”One file per process” and ”a single shared file”. In ”one file per process”, each process reads/writes data on its own file on disk and no coordination is required among processes. ”One single shared file” is more convenient to implement Parallel I/O, where all processes write to the same file on disk, but on different sections of that file. This method requires a shared file system that is accessible to all processes.

There are two ways in which multiple processes can access a shared file: independent access and collective access. In the first case, each process accesses the data directly from the file system without communicating or coordinating with the other processes. In collective access, small and fragmented accesses are combined into larger ones to the file system that helps significantly reduce access times. Our aim is to identify this kind of optimizations to explain the I/O behavior, for this reason, we propose a white box model.

B. Middleware

MPI is an interface and communications protocol used to program applications in parallel computers. It is designed to provide basic virtual topology, synchronization, and communication functionalities within a set of processes in an abstract way that is independent from the programming language used to develop the application.

MPI-IO functions work in similar way to those of MPI: writing MPI files is similar to sending MPI messages, and reading MPI files is like receiving MPI messages. MPI-IO also allows reading and writing files in a normal (blocking) mode, as well as asynchronously, to allow performing computation operations while the file on storage device is being read or written on the background. It also supports the concept of collective operations: each process can access MPI files on its own or all together, simultaneously. The second alternative

 ISBN: 978-1-61208-756-6

offers greater reading and writing optimizations that can be implemented on several levels. Mostly of MPI distribution provides MPI-IO functions by using ROMIO [11], which is an implementation of MPI-IO standard and it is used in MPI distributions, such as MPICH, MVAPICH, IBM PE and Intel MPI.

C. Parallel File Systems

A parallel file system is a distributed file system that stripes the files data into multiple data servers, connected to storage devices that provide concurrent access to the files through multiple tasks of a parallel application run on a cluster. The main advantages offered by a parallel file system include a global name space, scalability, and the ability to distribute large files through multiple storage nodes in a cluster environment, which makes a file system like this very appropriate for I/O subsystems in HPC. Typically, a parallel file system includes a metadata server with information about the data found on the data servers.

Some systems use a specific server for metadata, while others distribute the functionality of a metadata server through the data servers. Some examples of parallel file systems for high performance computing clusters are IBM Spectrum Scale, Lustre and PVFS2.

PVFS offers three interfaces through which PVFS files are accessed: PVFS' native Application Programming Interface (API), Linux kernel's interface, and ROMIO interface. The latter uses MPI to access PVFS files through MPI-IO's interface.

The underlying complexity of sending requests to all storage nodes and sorting file contents, among other tasks, is handled by PVFS. When a program attempts a reading operation on a file, small sections of the file are read from several storage devices in parallel. This reduces the load on any given disk controller and allows handling a larger number of requests.

D. Benchmarks

To evaluate the performance of parallel file system and test different I/O libraries of the I/O software stack, exists different I/O benchmark. Benchmarks are designed to mimic a specific type of workload in a component or system. One the most accepted I/O benchmark in HPC is IOR[12]. It supports several application I/O patterns and allows configuring them, and it offers access to shared files both independently and collectively. Additionally, IOR offers different execution options for the same algorithm using various parallel programming interfaces, including POSIX, MPI-IO, HDF5 and NETCDF.

III. FUNCTIONALITY ANALYSIS

To define an initial model of the I/O system, system functionality should be fully understood. First, the I/O pattern type to be analyzed was selected, and then the corresponding software stack layers for this model were applied. We have selected the IOR benchmark to evaluate I/O performance in HPC clusters. The analysis focused on the functionality that was observed for IOR in the data path.

Due to the heterogeneity of the I/O systems and the complexity of the software stack, an analysis was started for MPI-IO layers and the parallel file system. PVFS2 was the file system selected for our tests. At this time, we separated the different components considering the concepts of a parallel file system to allow us using the model with other parallel file systems, such as Lustre in the future.

The IOR benchmark offers the total runtime measurements for their programs, but they do not go into further detail in relation to the different abstraction layers of the parallel I/O system. These layers have to be crossed from the moment the user application sends an I/O request up until the CPU, through its operating system, effectively accesses the file on disk to read or write the data. Therefore, it is important to identify the layer in the software stack that requires more time during an I/O operation.

To follow the data path in the software stack, tracers or monitors can be used, but these operate on different levels of the I/O system. There is no single tool that allows recording the I/O behavior in all levels. For this reason, code instrumentation has been implemented in both the MPI-IO layer and the parallel file system layer.

A. Code Instrumentation

One possible way for finding out how the different modules in the I/O request process (application, middleware and file systems) work is instrumenting each of them by adding small sections of code. Thus, it would be possible to establish what percentage of the total runtime of an I/O operation corresponds to each of them, which would help knowing which of them is the most critical one and should be enhanced to dramatically improve parallel I/O speed.

Additional source code sections are simply a few lines of code written in C to monitor and measure runtime for some functions that were identified as critical during the request, service, and execution process of an I/O operation as it goes through each of the abstraction levels of the parallel system. To avoid hindering or interfering with the benchmark result screen printouts, the additional source code was added so that each process writes the local times of its own invocations to the critical functions of the parallel system to a local file on disk. Figure 2 shows the layers of the I/O system where code instrumentation was implemented. Left boxes in blue, green and orange represent the layers on compute nodes. The bigger orange box depicts the layers on the storage nodes. Small orange boxes represent the I/O clients, which interact with the metadata and data servers (storage nodes). We have measured the times for the different functions called by ROMIO and PVFS2.

B. Execution Environment

One of the problems found in production systems is that the file system cannot be modified and instrumented [13]. Therefore, to create the HPC cluster where the entire I/O software stack with the embedded code instrumentation will be installed, Amazon's EC2 platform service was used. This type of platforms offer various types of instances based on the type of service purchased. In this case, the cluster was deployed using the free service and, even though these nodes offer very limited functionality as regards number of CPUs, memory, storage and network; they proved to be adequate to create the necessary environment for the tests we needed.

 ISBN: 978-1-61208-756-6

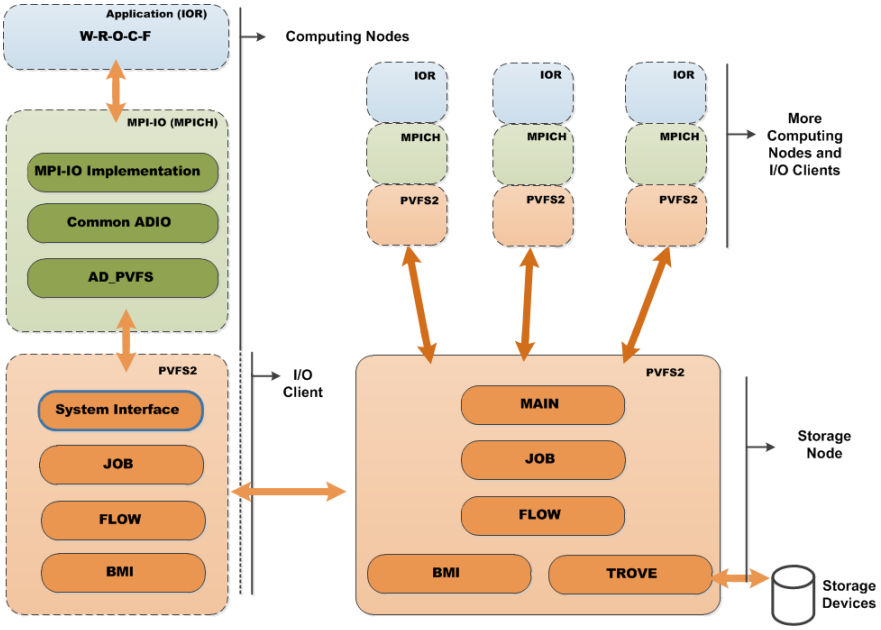


Figure 2: Code Instrumentation in the I/O Software Stack. Left boxes represent the layers on compute nodes and bigger orange box depicts the layers on the storage nodes.

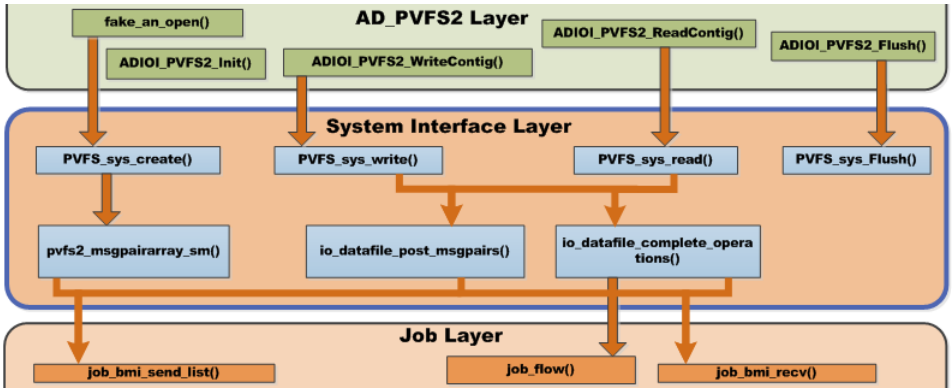


Figure 3: Selected functions of the System Interface layer.

Even though the execution environment affects the metrics obtained, the same configuration was run on a physical environment to compare it with that of Amazon. For instance, for a scenario with 8 computation nodes and I/O with one storage node, the times measured through code instrumentation in both environments are not the same, but they follow the same trend. The differences observed are mainly due to the different hardware performance in both execution environments.

Through the scenarios used, the critical functions involved in each layer of the I/O software stack were selected based on their role and execution time. As way of example, Figure 3 shows the functions selected in the System Interface layer of the I/O clients.

IV. MODELLING THE I/O SYSTEM

After analyzing each of the layers, a model of the I/O system was developed by implementing state machines and variables that describe each of those states. To that end, state machines were implemented for each of the layers in the system, differentiating their operation both on client and server side. The ultimate goal is using these state machines to design the behavior of each of the agents and its interactions with other agents and/or its environment.

The model developed is aimed to reproduce the interaction among the different components and analyzing how the information goes through the different modules or layers, with the possibility of measuring time to approach the real model of the I/O system. Therefore, each layer is modeled based on the execution flow of the functions that are called while processing certain requests, such as opening, closing, reading and writing operations. With the description of each function, the different states of the layers while carrying out those requests were implemented.

Due to the complexity to describe fully the modeling of the I/O software stack, we have selected the System Interface layer to explain in detail the calibration, verification and validation phases. Similar steps were done for the other layers.

The System Interface layer is a client-side interface that allows manipulating the objects in the file system. It launches a number of functions and state machines that process the operation in small steps.

 ISBN: 978-1-61208-756-6

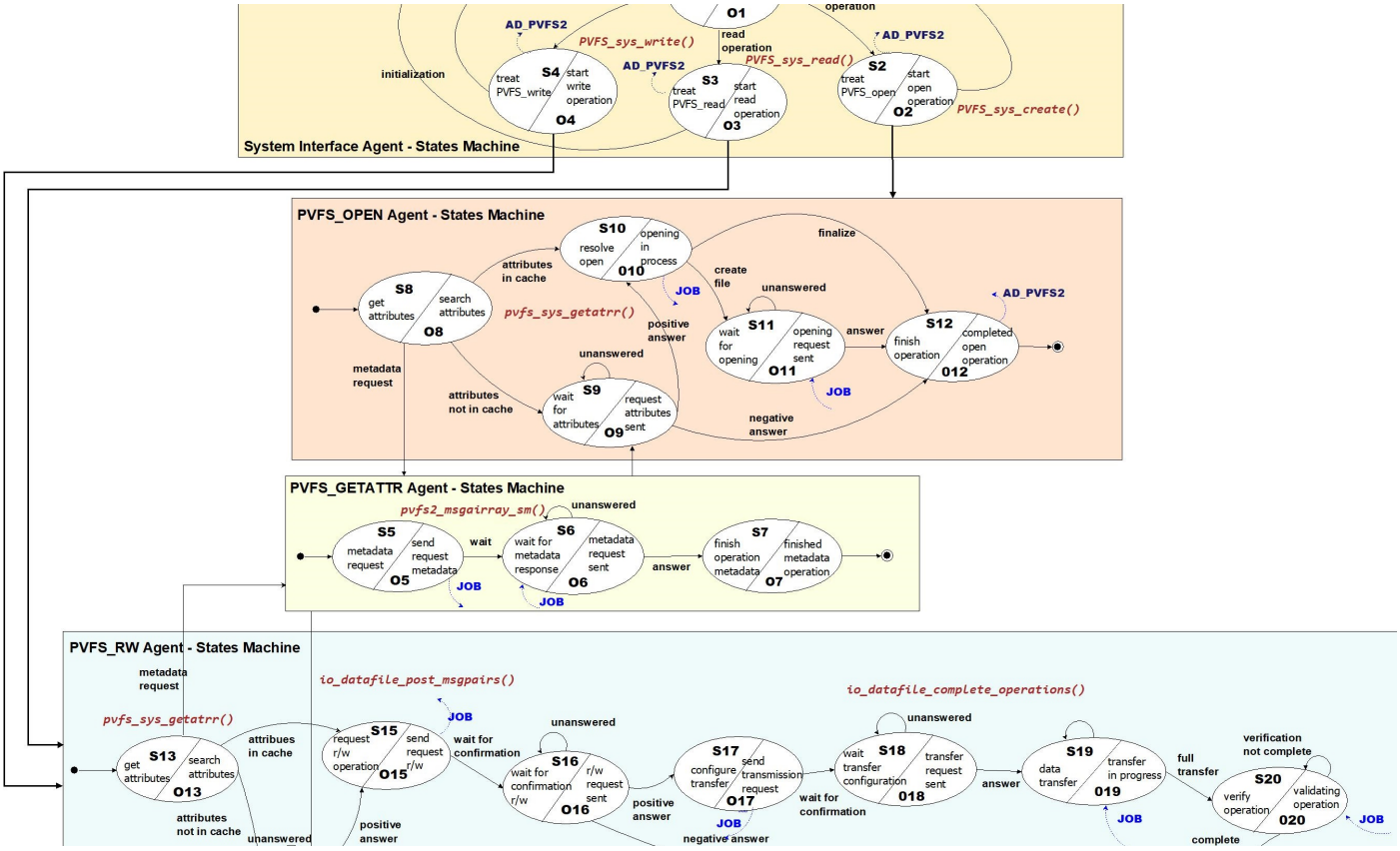


Figure 4: State machines for agents in the system interface layer.

In the context of PVFS, state machines execute a specific function in each of their states. The value returned by this function determines the state that should be adopted. Complex requests can be modeled; they are represented as a sequence of several states. Also, state machines can be nested to model and simplify common subprocess handling. These machines are used both in clients and servers.

There are several caches on the client side that are part of the System Interface layer and try to minimize the number of requests that the server has to process. The attributes cache (acache) manages metadata, while the name cache (ncache) stores the filename of file system objects and their respective handling number. To prevent caches from storing invalid information, data are set as invalid after a certain time has passed or when the server notifies the client that the object does not exist.

A. Functional Model

As shown in Figure 3, the functions in this layer are: `PVFS_sys_create()` to manage the creation of new files in the system, `PVFS_sys_write()` to perform writing operations, `PVFS_sys_read()` to perform reading operations, and `PVFS_sys_flush()` to dump file data to server storage. Each of these functions has internal variables and state machines that are run to carry out the relevant operations.

To simplify the model, we considered the following in relation to parallelism when handling several instances: **a)** I/O interfaces: layers `MPI-IO`, `ADIO` and `AD_PVFS` work in a sequential and blocking manner, since they run functions that require synchronization; this means that no instruction is served until the instruction being processed is completed. The calls run on their state machines are blocking; **b)** PVFS2 parallel file system: The System Interface, Job, Flow, BMI, Main Loop and Trove layers serve other requests and store their instructions in a buffer. Therefore, it allows handling different data flows.

The behavior of each of the agents is described by the state machine, the state transition table and the corresponding state variables. Figure 4 shows part of the state machines developed to model the operation of the System Interface layer, considering the functions and state machines corresponding to each of the three initial operations. As it can be seen, it consists of four agents called System Interface, which is responsible for decoding the instructions that enter the layer; `PVFS_OPEN`, which manages file opening operations; `PVFS_GETATTR`, which carries out searches in the metadata; and `PVFS_RW`, which manages file reading and writing operations.

The agent that manages file opening operations can only have one of five different states (S8 to S12). It will remain in S8 and configure agent `PVFS_GETATTR` if it requests metadata. If the attributes are not found in cache, it will transition to state S9 to wait for them; otherwise, it will transition to state S10. If in state S9, it will wait for a response from agent `PVFS_GETATTR` or it will complete the opening operation by communicating with the server, transitioning to state S10. If the operation cannot be completed, it will transition to state S12 to end.

While in state S10, it will start file creation through a request sent to the JOB layer, transitioning to state S11. Otherwise, it finishes the operation and transitions to state S12.

Figure 5: Simulator's user interface in NetLogo

While in state S11, it waits for a response to its file opening request and, if it receives one, it transitions to state S12. Once in state S12, it finishes the operation and sends a response to agent `AD_PVFS`.

On the other hand, agent `PVFS_RW` manages the write or read requests on client side. In Figure 4, there can be seen in red the functions selected that were used as the basis for the development of each state machine. For example, one of the functions belonging to `pvfs2_msgpairarray_sm()` [14], on which the `PVFS_RW` agent is based, is `io_datafile_post_msgpairs()` that is responsible for managing the data transmissions involved in the creation of files in agent System Interface. These communications occur, in the case of both a reading or writing, between client and server through the Job and BMI layers.

B. Initial model calibration

To obtain initial values for the functional model, we have monitored the selected functions for the IOR benchmark in a HPC cluster deployed in AWS EC2. The I/O system was configured over on PVFS2 parallel file system and the MPICH distribution. The cluster was composed by five nodes, where each one had three roles: compute node (computing and PVFS2 clients), metadata server and data server (datafiles). We have selected a simple pattern where file size and transfer size were updated. IOR was configured as follows:

- 1 GiB === `mpirun -np 5 ./ior -a MPIIO -b 205m -t 205m -F`
- 2 GiB === `mpirun -np 5 ./ior -a MPIIO -b 410m -t 410m -F`

For this setting, each process writes/reads to/from its own file in transfer sizes defined by the `-t` parameter. Due to the block size (`-b`) is equal to the transfer size (`-t`), only one operation is done by each process. The interface selected was MPI-IO for the *one file per process* (`-F`) strategy and independent I/O. The mapping corresponds to one MPI process per compute node.

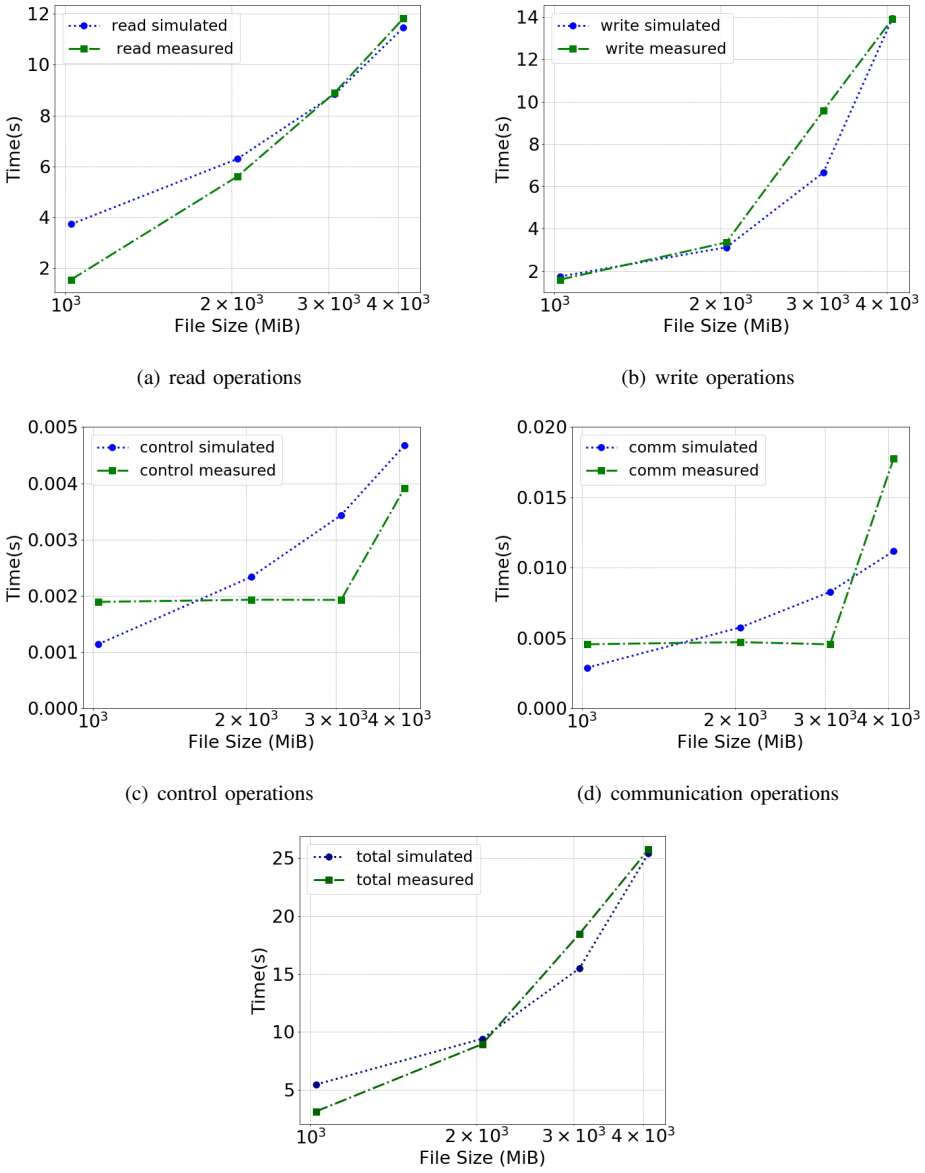
This measurement allows us to classify the monitored metrics in three groups: 1) *data access time* related with the data accesses operations such as write, read, and so on, 2) *control time* that includes verification and configuration of the data structures and 3) *communication time* related with the interaction between the clients and the metadata and data servers.

We have applied linear and exponential regressions for the time monitored in different functions of PVFS2. For this first analysis, we have selected as dependent variable the execution time and as independent variable the file size, request size is fixed for all the tests. In the case of the system interface layer, we have selected the following equations to represent the time of the functions:

- `PVFS_sys_create()` $= 0.0217 \times x$
- `PVFS_sys_write()` $= 0.8 \times e^{(0.7105 \times x)}$
- `PVFS_sys_read()` $= 2.5490 \times x + 1.2$
- `io_datafile_post_msgpairs()` $= 0.0012 \times x$
- `io_datafile_complete_operations()` $= 0.0028 \times x$

Where the x variable represents the file size to write or read. The statistical dispersion also depends on the file size and therefore it is calculated by using the same method.

 ISBN: 978-1-61208-756-6



(a) read operations

(b) write operations

(c) control operations

(d) communication operations

(e) total time

Figure 6: Simulated and Measured time in the system interface layer of the PVFS2

C. Initial Implementation

For the first proof of concept, an initial simulation model was developed using ABMS' framework called NetLogo. This framework includes a simplified programming language and a graphical interface that allows the user build, observe and use agent-based modeling without understanding complex standard programming language details. This tool is specifically indicated for the simulation of complex systems; it allows giving instructions to many independent agents that are concurrently executed, which is useful to study the connection between individual and collective behavior through agent actions and interactions.

The scenario adopted for the initial experiments was simulating the exchange of information among computation nodes and storage nodes, considering in each of them the layers discussed in previous sections. The MPI operations that can be served by the application layer are only I/O operations, and this initial implementation only includes open, read, write and close operations. One of the parameters allows toggling between executing only one type of operation or all of them. There is an option for selecting a maximum number of operations, which are distributed among the computation nodes selected.

The number of computation nodes and storage nodes can be configured. Node actions and interactions were fully implemented for the operations mentioned above. There are other parameters that allow selecting the existence of the data in the system before running the simulation, configuring the corresponding layers and preparing the I/O server for this scenario. Figure 5 shows the simulator's user interface. The configuration bars that the user has available to set the variables and parameters of the I/O software stack and the scenario to simulate are on the left. Also, the I/O configuration can be

 ISBN: 978-1-61208-756-6

made through command line. The center shows the distribution of the I/O system.

D. Verification and Validation

To validate the proposed model, we have configured a cluster in AWS EC2 similar to deployed in the calibration phase (see Section IV-B). The I/O system was deployed by using the PVFS2 parallel file system in a HPC cluster composed by five nodes, where each one was compute node (computing and PVFS2 clients), metadata server and data server (datafiles). IOR was executed for the following configurations:

- 1 GiB === `mpirun -np 5 ./ior -a MPIIO -b 205m -t 205m -F`
- 2 GiB === `mpirun -np 5 ./ior -a MPIIO -b 410m -t 410m -F`
- 3 GiB === `mpirun -np 5 ./ior -a MPIIO -b 615m -t 615m -F`
- 4 GiB === `mpirun -np 5 ./ior -a MPIIO -b 820m -t 820m -F`

Figure 6 presents the simulated and measured times for the IOR benchmark in the system interface layer of the PVFS2. As can be seen, the I/O behavior in this layer is dominated by the access data operations that corresponds to the read and write operations. The simulated total time of System Interface layer shows similar behavior to measured time but we can see that the control and communication time present a higher deviation for more than 3 GiB. However, due to the data access operations represent the highest I/O time we can consider the initial simulation model appropriate to represent the I/O behavior.

About the strange behavior of the communication and control functions, it can be observed in Figure 6(c) and 6(d) that these are constant for file size up to 3 GiB but these grows up for 4 GiB. We have also modeled the communication and control times by using the constant and exponential functions but these do not fix with the right behavior for the cases evaluated. Considering the evaluated pattern, these functions did not impact significantly on the I/O behavior but we must evaluate other file sizes and I/O patterns to can reduce the deviation and guarantee that these functions will have not impact on the I/O behavior independently of the data access pattern.

V. CONCLUSIONS

This paper presented a conceptual model of HPC I/O software stack by using agents based on state machines that act and communicate within the defined environments.

The instrumentation method used to obtain the different parameters for the simulator has been described. Likewise, the functionality found through the instrumentation of the system code was useful for the generation of state machines. This methodology allowed us to study the working of the system without the difficulty of obtaining exhaustive descriptions of the system, required by other modeling paradigms. An initial implementation using ABMS with NetLogo was validated for the IOR benchmark configured for sequential pattern, a single shared file, MPI-IO interface and independent I/O.

As future work, we will continue analyzing other techniques to monitor the data transfer rate (bandwidth) and the input/output operations per second (IOPs) at different levels of the I/O software stack. Furthermore, we will evaluate collective operations and other I/O strategies. Additionally, we will extend the model for the Lustre parallel file system.

ACKNOWLEDGMENT

This research has been supported by the Agencia Estatal de Investigación (AEI), Spain and the Fondo Europeo de Desarrollo Regional (FEDER) UE, under contract TIN2017-84875-P and partially funded by the Fundacion Escuelas Universitarias Gimbernat (EUG).

We thank Román Bond, research engineering of Universidad Nacional Arturo Jauretche (Argentina), for his support in the implementation of the simulator.

REFERENCES

[1] A. Núñez, J. Fernández, J. D. Garcia, F. Garcia, and J. Carretero, "New techniques for simulating high performance mpi applications on large storage net," J. Supercomput., vol. 51, no. 1, Jan. 2010, pp. 40–57.

[2] J. Kunkel, "Using Simulation to Validate Performance of MPI(-IO) Implementations," in Supercomputing, ser. Lecture Notes in Computer Science, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds., no. 7905. Berlin, Heidelberg: Springer, 06 2013, pp. 181–195.

[3] N. Liu et al., "Modeling a leadership-scale storage system." in PPAM (1), ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 7203. Springer, 2011, pp. 10–19.

[4] B. Feng, N. Liu, S. He, and X.-H. Sun, "HPIS3: Towards a High-performance Simulator for Hybrid Parallel I/O and Storage Systems," in Proceedings of the 9th Parallel Data Storage Workshop, ser. PDSW '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 37–42.

[5] C. Carothers, D. Bauer, and S. Pearce, "ROSS: a high-performance, low memory, modular time warp system," in Fourteenth Workshop on Parallel and Distributed Simulation., 2000, pp. 53–60.

[6] V. J. Julián and V. J. Botti, "Estudio de metodos de desarrollo de sistemas multiagente," Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial, vol. 7, 2003, pp. 65–80. [Online]. Available: http://www.redalyc.org/articulo.oa?id=92501806. Retrieve: 10/2019

[7] A. Borshchev and A. Filippov, "From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools," The 22nd International Conference of the System Dynamics Society, Oxford, England, 07 2004.

[8] E. Kremers, "Modelling and Simulation of Electrical Energy Systems through a Complex Systems Approach using Agent-Based Models," Ph.D. dissertation, Universidad del País Vasco (UPV/EHU), 2012.

[9] M. Taboada, E. Cabrera, F. Epelde, and E. Luque, "Using an agent-based simulation for predicting the effects of patients derivation policies in emergency departments," in International Conference on Computational Science, Barcelona, Spain, 2013, pp. 641–650.

[10] Sharcnet. Parallel I/O introductory tutorial. [Online]. Available: https://www.sharcnet.ca/help/index.php/Parallel_IO_introductory_tutorial. Retrieve: 10/2019. (2017)

[11] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–. [Online]. Available: http://dl.acm.org/citation.cfm?id=796733. Retrieve: 10/2019

[12] T. M. William Loewe and C. Morrone. IOR Benchmark. [Online]. Available: https://github.com/chaos/ior/blob/master/doc/USER_GUIDE. Retrieve: 10/2019. (2013)

[13] P. Gomez-Sanchez et al., "Using AWS EC2 as Test-Bed infrastructure in the I/O system configuration for HPC applications," Journal of Computer Science & Technology, vol. Volumen 16, no. 02, pp. 65-75, 11/2016 2016. [Online]. Available: http://hdl.handle.net/10915/57264. Retrieve: 10/2019

[14] PVFS2 Team, "PVFS 2 File System Semantics Document," PVFS Development Team, Tech. Rep., 2015.

 ISBN: 978-1-61208-756-6