A High-Level Interpreted MPI Library for Parallel Computing in Volunteer Environments

Troy P. LeBlanc Jaspal Subhlok Edgar Gabriel

Department of Computer Science
University of Houston
Houston, TX 77204

***Abstract*— Idle desktops have been successfully used to run sequential and master-slave task parallel codes on a large scale in the context of volunteer computing. However, execution of message passing parallel programs in such environments is challenging because a pool of nodes to execute an application may have architectural and operating system heterogeneity, can include widely distributed nodes across security domains, and nodes may become unavailable for computation frequently and without warning. The VolPEx (Parallel Execution on Volatile Nodes) tool set is building MPI support in such environments based on selective use of process redundancy and message logging. However, addressing this challenge requires tradeoffs between performance, portability, and usability. The paper introduces a robust MPI library that is designed to be highly portable across heterogeneous architectures and operating systems. This VolpexPyMPI[1] library is built with Python, works with Linux and Windows platforms and accepts user level MPI programs written in C or FORTRAN. The performance of VolpexPyMPI is compared with a traditional C based implementation of MPI. The paper examines in detail the tradeoffs of these usability focused and performance focused approaches.**

I. Introduction

To date, most desktop computers and workstations are virtually idle as much as 90% of the time, representing what the volunteer computing community sees as an attractive "free" platform for data parallel computations. Idle desktops have been successfully used to run sequential and master-slave task parallel codes, most notably under Condor [12] and BOINC [1]. Extending the classes of application that can be executed in a volunteer environment turns out to be highly challenging, since the compute resources are heterogeneous, have varying compute, memory and network capacity, and become unavailable for computation frequently and without warning. Further, the nodes are connected with a shared network where available latency and available bandwidth can vary. Because of these properties, we refer to such nodes as *volatile* and parallel computing on volatile nodes is the focus of this work.

Traditionally, MPI libraries such as OpenMPI [6] are implemented in C/C++ and use, in the case of Ethernet networks, non-blocking TCP sockets for communication. While this might be the best approach from the performance perspective, porting such an implementation to non Unix/Linux based operating systems – most notably Windows – and supporting arbitrary configurations of various operating systems and architectures is a non-trivial task that requires significant effort. The main reason is that system calls are not necessarily identical on different operating systems, and complex configuration scripts are often required as glue for a heterogeneous environment. The same holds for supporting heterogeneous hardware configurations, especially when the data representation supported by the processors differs, requiring potentially byte swapping and data padding for each element of a message. Furthermore, it has been documented that for utilizing systems with strongly varying performance, an internal flow control is required within the MPI library to make sure that a slow node is not flooded with data from a fast node [4].

In this paper we explore an alternative approach to provide message passing communication in volunteer computing environments based on Python, a high-level programming language. Python offers a number of features that make it highly attractive to volunteer computing environments such as platform independence due to the fact that it is an interpreted language; seamless handling of Linux, Windows or other operating systems as well as any combination of them; and a large collection of available libraries which ease the development of Python based applications and systems. In contrast to some existing Python MPI libraries, our library targets volunteer computing environments and supports process failures.

VolpexPyMPI consists of two major building blocks. The first is a node selection framework to monitor a set of volunteer computers that could be tasked with executing a standard MPI program. The second building block is the actual MPI library. The MPI library utilizes the Python/C Application Programming Interface (API) to interface the VolpexPyMPI library to execute unaltered MPI programs written in C or FORTRAN. The resulting library system allows execution on a mix of Linux and Windows nodes in a framework that can execute unaltered user MPI programs.

The paper is organized as follows. Section III presents the design of the node selection framework and the MPI library. Section IV analyzes the performance of the VolpexPyMPI, with an overall discussion of the findings in section V. Section II presents the related work in the area and finally section VI contains conclusions.

[1]This work was conducted as a part of the author's doctoral research while on Fellowship from the NASA Johnson Space Center.

II. Design and implementation

VolpexPyMPI is an MPI library targeting volunteer computing environments. In order to provide reliable execution on failure prone volunteer computing environments, the Volpex approach to MPI library design is centered around the following principles:

1) *Controlled redundancy:* A process can be initiated as two (or more) replicas. The execution model is designed such that the application progresses at the speed of the fastest replica of each process, and is unaffected by the failure or slowdown of other replicas.
2) *Receiver based direct communication:* The communication framework supports direct node to node communication with a *pull* model: the sending processes buffer data objects locally and receiving processes contact one of the replicas of the sending process to get the data object.
3) *Distributed sender based logging:* Messages sent are implicitly logged at the sender and are available for delivery to process instances that are lagging due to slow execution.

As stated earlier, VolpexPyMPI comprises two major building blocks, namely the *node selection framework* and the `MPI Library`. Among the foremost goals of VolpexPyMPI is the ability to execute on Linux and Windows nodes over a Wide Area Network (WAN) and organize communication between replicated sets of nodes. Furthermore, it must also be able to organize data subset delivery to nodes, and track availability of nodes. Finally, it must be able to deliver and execute unaltered MPI programs on the nodes.

A. Node Selection Framework

The main goal of the node selection framework is to monitor a set of volunteer computers that could be tasked with executing a parallel application. The VolpexPyMPI node selection framework was built as an asynchronous XML-RPC webserver (henchforth called Volpex server) and XML-RPC clients (henthforth called Volpex clients). The Volpex server and client software were written in Python and can run on Linux and Windows systems.

Among the main functionality of the Volpex server is to maintain a GlobalMap, which is a configuration file in tabular representation showing the status of all Volpex clients. The GlobalMap is implemented using Python's SQLite3 module, a lightweight disk-based database that does not require a separate server process. The GlobalMap status, includes the last check-in time (reported as DDD/HH:MM:SS), the last status message from the server to the client, the last status message from the client to the server, and the type of OS running on the client node. The Volpex server status messages include abort, start, inactive, suspect, and the Volpex client status messages include done, active, ready and avail. Upon start of an MPI application, the Volpex server forwards the GlobalMap to the Volpex clients, which store it as an in-memory data structure.

The Volpex server utilizes port 8080 and is run on a standard Linux server also running the Apache webserver which utilizes port 80 for user interaction in MPI run setup. The Volpex server interacts with the Volpex clients over the specified port and through a set of public functions described later. The user interface controls available to the end user are via CGI scripts used to interact with the Volpex server via a webpage. The webpage allows to set the number of MPI processes, the level of redundancy required (currently limited to single, double, or triple redundancy), and upload the user's MPI program to the server for compilation. For the purposes of initial testing an additional drop-down list interface allowed easy selection, upload and compilation of the NAS Parallel Benchmarks. The user interface also has other buttons such as Execute, Print Event Log and Reset Mapping for controlling the test case execution.

The webpage interface also shows the *SendBuffer* and the *Event Log* on a per process basis. The role of the SendBuffer is detailed in the following subsection. The Event Log offers the ability to track the progress of each process individually. Note that the event log is typically turned off for actual performance measurements. Each node in the GlobalMap that is involved in the execution of a particular MPI run can post the contents of its SendBuffer and Event Log to the Volpex server after the run for analysis.

Performance monitoring of the Volpex server and clients shows a relatively low CPU utilization of about 0.1% during quiescent timeframes. CPU and memory utilization during MPI runs will, of course, increase dramatically.

B. MPI Library Functions

As mentioned earlier the foremost goal of the MPI library on VolpexPyMPI is to ease the support of heterogeneous software and hardware configuration, ease the deployment of the applications onto the client nodes, and provide a robust communication environment between the client nodes. From the conceptual perspective, the main features of the VolpexPyMPI library are (i) the ability to execute multiple copies of each MPI process and (ii) utilize a receiver based *pull* model combined with sender-side message logging. In the following we detail both aspects.

To ensure failsafe completion of MPI programs, the Volpex XML-RPC server builds a configuration file, the GlobalMap, described in the previous section. The configuration file is passed to the Volpex clients and allows identification of redundant independent nodes. A new naming scheme that allows identification of redundant nodes is established and assigned when the configuration file is read into memory. The scheme uses the normal rank identifier used in other MPI installations and as well as a letter to identify redundancy level. As shown in Figure 2, node "0,A" indicates the rank 0 of the first redundancy level whereas, for example, "3,C" indicates the rank 3 node of the third redundancy level. It should be noted that VolpexPyMPI assigns every process to a different processor and that the MPI application is not aware of the redundancy level of any process.

Fig. 1. VolpexPyMPI employs redundant nodes to provide fault tolerance in the volunteer environment.

The VolpexMPI Library uses a *pull* model, as described previously to accomplish using the redundant nodes to successfully complete a MPI program run even if one or more volunteer nodes fail to complete execution. To implement this functionality in VolpexPyMPI, the following functional requirements are met:

1) `MPI_Send` and `MPI_Isend` are copying data to a local SendBuffer such that any node or redundant copy can request to receive a message from any other node or redundant copy.
2) `MPI_Recv` and `MPI_Irecv` first send a short message to check if a connection to a source node can be made before attempting to receive the real message.
3) Each redundant group of nodes operates independently. If no failures or drop-offs occur then all redundant groups would successfully complete.

Since different replicas can be in different execution states, a message matching scheme has to be employed to identify which message is being requested by a receiver. For deterministic execution, a simple scheme that timestamps messages by counting the number of messages exchanged between pairs of processes is applied based on the tuple [communicator id, message tag, sender rank, receiver rank]. These timestamps are also used to monitor the progress of individual process replicas for resource management. Furthermore, a late replica can retrieve an older message with a matching logical timestamp, which allows restart of a process from a checkpoint.

The SendBuffer provides the functionality to store and retrieve an MPI message based on the tuple described above. An important question is whether the message buffers on the sender processes must be maintained for the duration of execution or whether they can be cleared at some point. From the logical perspective, a message buffer can never be cleared due to the fact that, even if all replicas of a particular rank have received a given message, all of them might fail to finish the execution. Thus, a new replica of that process might have to be started, which would have to retrieve all messages. There are two versions of the Sendbuffer management available in VolpexPyMPI. The first version utilizes once again the SQLite3 module and does not limit as of today the size of the SendBuffer. The second version uses a Python array treated as a circular buffer. Note, that the long-term goal is to coordinate the size of the SendBuffer with checkpoints of individual processes, which will allow guaranteed restarts with a bounded buffer size.

For the actual data transfer between the MPI processes, VolpexPyMPI has once again two options: the first utilizes the XML-RPC services for communication between the processes. The second version uses a standard Python Threaded TCP SocketServer class for the direct node-to-node communication, and uses the XML-RPC service only for the communication between to the Volpex server and clients.

TABLE I
MPI FUNCTIONS IMPLEMENTED IN VOLPEXMPI TO EXECUTE THE NPBS

MPI_Init	MPI_Finalize
MPI_Send	MPI_Recv
MPI_Isend	MPI_Irecv
MPI_Reduce	MPI_Allreduce
MPI_Alltoall	MPI_Alltoallv
MPI_Wait	MPI_Waitall
MPI_Comm_rank	MPI_Comm_size
MPI_Bcast	MPI_Barrier
MPI_Comm_split	MPI_Comm_dup
MPI_Abort	MPI_Wtime

Figure **??** shows the VolpexPyMPI architecture and Table I lists all of the MPI functions chosen for this first version of VolpexPyMPI which is based on their usage in the NAS Parallel Benchmarks.

Fig. 2. VolpexPyMPI Architecture

In the following, we describe the four most basic functions of the library, namely `MPI_Init, MPI_Finalize, MPI_Send` and `MPI_Recv`. All other functions are, in fact, built upon these four functions.

`MPI_Init` initializes the VolpexPyMPI execution environment. Among the items initialized are the Python runtime environment, a request list for the management of non-blocking messages, structures for tracking the logical time stamps, reading the configuration file into a local data structure, initializing the SendBuffer, and resolving the private IP address and hostname of the local processor. Each process furthermore reads the GlobalMap from the Volpex server and appends IP addresses and ports to the active in-memory data structures.

`MPI_Finalize` terminates the VolpexPyMPI execution environment and frees data structures. `MPI_Finalize` first waits for a predefined amount of time (e.g. 5 seconds). This additional waiting time allows other processes to still retrieve messages and finish gracefully for benchmarking purposes, but is not required for production runs later on. Using `Py_Finalize`, a Python library call, VolpexPyMPI shuts down the Python run-time interpreter which will finally de-allocate the data structures such as the SendBuffer.

`MPI_Recv` highlights how the receiver based direct communication works; therefore, we would like to discuss it in some more details. First, we determine the logical timestamp of the message based on the tuple of [sender rank, receiver rank, communicator and tag]. Next a `TargetSelect` function is called to determine which target nodes to call and in which order to call them in case of failures. For example, if a process with the rank 1 wants to receive a message from the process with rank 2 and the run was utilizing triple redundancy, the `TargetSelect` function would read the GlobalMap for the particular communicator to determine the IP address and port number of nodes ”2,A”, ”2,B” and ”2,C”. The MPI function would then proceed in a round-robin fashion attempting to receive the message. This usually starts with an attempt to contact the rank 2 within the same redundancy level as the

requesting rank 1 node. If the node is unreachable, the MPI receive function proceeds to the next higher redundancy level or loops around to the lowest level, if necessary. If the first receive call is successful, the MPI library proceeds normally without ever attempting to contact the redundant nodes. If attempts are unsuccessful, the target is marked as unreachable within the GlobalMap and there are no further attempts to contact the node for data. Once a target is chosen, a TCP call is made to the target MPI process' TCP service via the VolpexSBRequest.

Note, that the non-blocking function `MPI_Irecv` capitalizes on the use of Threads and allows a request for data to proceed in the background until the associated MPI_Wait request is encountered.

The implementation of `MPI_Send` (and `MPI_Isend`) in VolpexPyMPI is fairly simple, since it only copies data to the local SendBuffer after the size of the MPI datatype used is determined. The library also determines the logical time stamp of each message and stores it along with the other communication parameters in order to uniquely identify a message upon request from a receiver process.

III. Performance Evaluation

The evaluation focuses on characterizing the performance of the Python MPI Library. And, while it would be interesting to evaluate the performance of the MPI Library on heterogeneous volunteer computers, we executed benchmarks on a dedicated cluster in order to have a controlled test environment and to have comparable performance results for an interpreted language implementation. The dedicated cluster utilizes 29 compute nodes, 24 of them having a 2.2 GHz dual core AMD Opteron processor, and 5 nodes having two 2.2GHz quad-core AMD Opteron processors. Each node has 1 GB main memory per core and network connected by 4xInfiniBand as well as a 48 port Linksys GE switch. We utilized Python 2.4 across the cluster. For evaluation we utilize the Gigabit Ethernet network interconnect of the cluster to compare the VolpexPyMPI run times to the C-version of VolpexMPI [9].

The BT, CG, EP, IS and SP benchmarks from the NAS Parallel Benchmarks suite are executed for 4 process counts and a data class set size of S. For each experiment, the run times were captured as established and reported in the NPB with the normal `MPI_Wtime` function calls for start and stop times.

Figure 4 shows the first test results for runs of 4 processes utilizing the Class S data sets for the five NPBs of interest. These reference executions did not employ redundancy. The run times for the C version of VolpexMPI are shown for comparison in the bar graph. All times are noted as normalized execution times with a reference time of 100 for the C version of VolpexMPI. Two different sets of results are shown for VolpexPyMPI. Version 1 contains the SQLite3 version of the SendBuffer management, and uses the XML-RPC service for node-to-node communication. Version 2 uses the circular buffer for SendBuffer management and the threaded TCP-service for data transfer. The results indicate, that version 1 of VolpexPyMPI run times are between 5 and 55 times greater than the corresponding C version of the library. VolpexPyMPI version 2 greatly improves the execution time of these benchmarks, reducing the overhead to a more manageable factor of 1.5 in the best case and 6.5 in the worst case compared to the C-only counterpart. Therefore, for the rest of the analysis we are focusing entirely on the 2nd version of VolpexPyMPI.

Fig. 3. Performance results of two versions of VolpexPyMPI compared to the C version.

Next, we document the effect of executing an application with multiple copies of each MPI process. The left part of Figure 5 shows the normalized execution times of VolpexPyMPI for the very same test cases running with no (same as single) redundancy (x1), double redundancy (x2) and triple redundancy (x3). The results indicate that, for most benchmarks, the overhead due to redundant execution is minimal if no failure occurs, i.e. executing multiple copies of each MPI processes does not impose a significant performance penalty in the VolpexPyMPI scheme/model. The double redundancy (x2) runs show minimal sensitivity to process replication; however, the triple redundancy (x3) runs show higher sensitivity to process replication, especially for BT and SP. This is possibly due to the fact that requests for data are made in a round-robin fashion to all redundant processes where the requesting process waits only for a short time and then attempts to call another source target.

Fig. 4. Performance results of VolpexPyMPI for single, double and triple redundancy.

Finally, we document the performance impact of a process failure for the NAS Parallel Benchmarks when using VolpexPyMPI. For this, we inserted into the source code of each benchmark some statements which terminate the execution of the second replica of rank 1 in `MPI_COMM_WORLD`, emulating a process failure. All processes communicating with the terminated process will thus have to repost all pending communication operations to the only remaining replica of process 1. This test case represents one of the worst case scenarios for VolpexPyMPI, since the number of processes communicating with a single process doubles at runtime. Killing more than one process would actually relieve the remaining processes with rank 1, since the number of communication partners is reduced. The results shown in Figure 6 show virtually no overhead in the scenario outlined above compared to the fault-free execution of the same benchmark using double redundancy.

Fig. 5. Performance results of VolpexPyMPI in case of a process failure.

IV. Discussion

Our experiments demonstrate, that unaltered MPI applications such as given with the NAS Parallel Benchmarks

can be executed using VolpexPyMPI. In our experience, the fundamental benefits of using a 'Python only' approach including components such as SQLite3 and the XML-RPC scheme are the increased portability of the software, ease of development of the MPI library and ease of deployment of applications. The consequences of this approach are, however, a significant performance degradation compared to using a 'legacy' programming language such as C. By combining both approaches, namely using Python for the overall management of processes and non-performance critical section, and using more traditional TCP services and in-memory data structures for performance critical aspects of the library, we managed to create a hybrid implementation that combines the best of both worlds: portability of python and a manageable performance overhead. Finally, the applications that are likely to be executed in a volunteer environment are not likely to be as communication intensive as most NAS benchmarks. Hence, the overheads of NAS benchmarks can be considered to be a worst case scenario.

V. Related Work

Fault tolerant methods for MPI libraries can be divided into two main categories: approaches using check-point restart mechanism, and libraries using replication techniques.

There are a large number of MPI libraries that incorporate checkpoint-restart for fault-tolerance, with MPICH-V [3] being probably the best known example. This library is based on uncoordinated check-pointing and pessimistic message logging. The library stores all communications of the system on reliable media through the usage of a *channel memory*. In case of a process failure, MPICH-V is capable of restarting the failed application process from the last checkpoint and replay all messages to that process. Although some of the conceptual aspects of our work are similar to MPICH-V, there are key architectural differences, most notably the ability to run multiple replicas of an MPI process.

MPI/FT [2] and P2P-MPI [7] are based on process replication techniques. In P2P-MPI for example, each set of process replicas maintain a master replica that distributes messages. Fault detection is done using a gossip-style protocol. P2P-MPI also takes advantage of locality awareness and co-allocation strategies. A key difference between P2P-MPI and the approach presented in this paper is the utilization of a *pull* model for data communication in VolpexPyMPI that allows execution to be driven by the fastest replica.

Furthermore, there are a number of Python based MPI libraries. Pypar [10] provides bindings to a subset of the message passing interface standard MPI. To execute MPI programs, Pypar depends on a Python installation, an MPI library, Numeric Python and a C compiler. Pypar works with each python process importing pypar which, in turn, imports a shared library that has been statically linked to the C MPI library (e.g., mpich). MPI for Python (mpi4py) [5] provides an object oriented approach designed for translating MPI syntax and semantics from C++ to Python. This implementation passes general Python objects for blocking point-to-point communications. Mpi4py depends on a working MPI distribution and Python 2.3 minimally. Pypar and Mpi4py are not pure implementations of the MPI standard.

The final two examples researched are pure Python implementations. The pyMPI [11] completely rebuilds the python interpreter to be an integral part of each parallel process. The intention of the developers is to provide an MPI library for MPI programs written in Python. This differs fundamentally from the basic requirement of VolpexPyMPI to be able to compile and execute user MPI programs written in both C and FORTRAN. Pydusa [8], also called MyMPI, is closest in design to VolpexPyMPI because it supports heterogeneity as well as dynamic process creation. It uses the existing Python interpreter and has implemented 30 MPI functions in its library. This implementation does not currently support non-blocking point-to-point communication or redundancy. Pydusa is also intended for executing MPI programs written in Python.

In conclusion, the researched Python implementations of MPI are mostly scoped to allow the user to write MPI programs in Python versus C or FORTRAN. While this may be a useful tool on some cases, the majority of the scientific MPI programs are written in C or FORTRAN.

VI. Conclusions

This paper explored a Python based implementation of MPI for volunteer computing environments. We demonstrated that the fundamental approach is appropriate to provide fault-tolerance based on process replication and a *pull* model communication scheme combined with sender based message logging. The ongoing work on VolpexMPI and VolpexPyMPI includes developments in algorithms and execution environments. We are working on integrating checkpoint-restart with VolpexMPI to dynamically manage replication by recreating slow and failed replicas from healthy replicas. We are also actively developing novel algorithms for target selection that involves identifying groups of process that are closest to each other in network distance at execution stage. Furthermore, we are currently testing the NAS benchmarks with larger process counts and investigating several scientific applications as candidates for execution on desktop computer clusters for VolpexMPI. Candidate applications have typically large memory and compute requirements combined with a low degree of communication between the individual MPI processes.

VII. Acknowledgments

Partial support for this work was provided by the National Science Foundation's Computer Systems Research program under Award No. CNS-0834750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] D. Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, November 2004.

[2] Rajanikanth Batchu, Jothi P. Neelamegam, Zhenqian Cui, Murali Beddhu, Anthony Skjellum, and Yoginder D. Mpi/ft tm : Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *In Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*, pages 26–33, 2001.

[3] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[4] IMPI Steering Committee. IMPI - interoperable message-passing interface. http://impi.nist.gov/.

[5] Lisandro Dalcin. Mpi for python. http://mpi4py.scipy.org/.

[6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[7] Stephane Genaud and Choopan Rattanapoka. Large-scale experiment of co-allocation strategies for peer-to-peer supercomputing in p2p-mpi. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium*, pages 1–8, 2008.

[8] Timothy H. Kaiser. Pydusa- parallel programming in python. http://sourceforge.net/projects/pydusa/, 2008.

[9] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. Volpexmpi:an mpi library for execution of parallel applications on volatile nodes. In *In Proc. The 16th EuroPVM/MPI 2009 Conference, Espoo, Finland*. Springer-Verlag LNCS,5759, September 2009.

[10] Ole Moller Nielsen. Pypar. http://datamining.anu.edu.au/ ole/pypar/.

[11] P. Miller. pyMPI An introduction to parallel Python using MPI. https://computing.llnl.gov/code/pdf/pyMPI.pdf, September 2002.

[12] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.