VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes

Troy LeBlanc, Rakhi Anand, Edgar Gabriel, and Jaspal Subhlok

Department of Computer Science, University of Houston
{tpleblan,rakhi,gabriel,jaspal}@cs.uh.edu

Abstract. The objective of this research is to convert ordinary idle PCs into virtual clusters for executing parallel applications. The paper introduces VolpexMPI that is designed to enable seamless forward application progress in the presence of frequent node failures as well as dynamically changing networks speeds and node execution speeds. Process replication is employed to provide robustness in such volatile environments. The central challenge in VolpexMPI design is to efficiently and automatically manage dynamically varying number of process replicas in different states of execution progress. The key fault tolerance technique employed is fully distributed sender based logging. The paper presents the design and a prototype implementation of VolpexMPI. Preliminary results validate that the overhead of providing robustness is modest for applications having a favorable ratio of communication to computation and a low degree of communication.

1 Introduction

Idle desktop computers represent an immense pool of unused computation, communication, and data storage capacity [1,2]. The advent of multi-core CPUs and increasing deployment of Gigabit capacity interconnects have made mainstream institutional networks an increasingly attractive platform for executing scientific codes as "guest" applications. Idle desktops have been successfully used to run sequential and master-slave task parallel codes, most notably under Condor [3] and BOINC [4]. In the recent past, some of the largest pools of commercial compute resources, specifically Amazon [5] and Google [6], have opened up part of their computation farms for public computing. Often these computers are very busy on a few occasions (e.g. Christmas shopping) and underutilized the rest of the time. This new phenomenon is often referred to as "cloud computing".

However, a very small fraction of idle PCs are used for such guest computing and the usage is largely limited to sequential and "bag of tasks" parallel applications. In particular, we are not aware of any MPI implementation that is used widely to support execution on idle desktops. Harnessing idle PCs for communicating parallel programs presents significant challenges. The nodes have varying compute, communication, and storage capacity and their availability can change frequently and without warning as a result of, say, a new host application, a reboot or shutdown, or just a user mouse click. 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 challenging.

This paper introduces VolPEx (Parallel Execution on Volatile Nodes) MPI that represents a comprehensive and scalable solution to execution of parallel scientific applications on virtual clusters composed of volatile ordinary PC nodes. The key features of our approach are the following:

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. (Replicas may also be formed by checkpoint based restart of potentially failed or slow processes, but this aspect is not implemented yet).
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 or recreation from a checkpoint.

VolpexMPI is designed for applications with moderate communication requirements and is expected to scale to 100s of nodes on institutional LANs. Certainly many parallel applications will not run effectively on ordinary desktops under VolpexMPI (or any other framework) because of memory and communication requirements that can only be met with dedicated clusters. In particular, for an application to run effectively on volatile nodes, it must have a low communication degree and limited sensitivity to latency. It has been shown that many scientific applications have a low degree stencil as the dominant communication pattern [7,8]. Hence, we believe that many parallel applications are good candidates and an important goal of this project is to identify the extent of applicability of this approach.

An example motivating application is Replica Exchange Molecular Dynamics (REMD) formulation [9] where each node runs a piece of molecular simulation at a different temperature using the AMBER program [10]. At certain time steps, communication occurs between neighboring nodes based on the Metropolis criterion, in case a given parameter is less than or equal to zero. REMD requires low volume loosely coupled communication making it a good candidate for VolpexMPI. It is currently implemented in the Volpex environment [11] but not yet ported to MPI.

This paper focuses on the design, implementation and validation of VolpexMPI. The implementation works on clusters and PC grids. Preliminary results presented for a commodity PC cluster compare VolpexMPI to Open MPI and analyze the overhead of replication and node failure.

2 Fault Tolerance in MPI

The MPI specifications are rather vague about failure scenarios. In recent years MPI implementations have been developed to deal with process and network failures. Fault tolerant methods supported by various implementations of MPI can be divided into three categories: 1) extending the semantics of MPI, 2) check-point restart mechanism, and 3) replication techniques.

FT-MPI [12] is the best known representative of the approach of extension of semantics for failure managment. The specification of FT-MPI defines the status of the MPI handles and messages in case of a process failure. FT-MPI has the ability to either replace a failed process, or continue execution without it. The library deals only with MPI-level recovery, but lets the application manage the recovery of user level data items in a performance efficient manner [13], However, it requires significant modifications to the application, and thus does not provide a transparent fault-tolerance mechanism.

MPICH-V [14] belongs to the category of MPI libraries that employ checkpoint-restart mechanisms for fault tolerance. It 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. Similarly to MPICH-V, RADICMPI [15] also fundamentally relies on checkpointing MPI processes, but tries to avoid any central instance or single point of failure within its overall design. Although some of the conceptual aspects of VolpexMPI are similar to MPICH-V and RADICMPI, there are key architectural differences. Neither of these two MPI libraries are designed to run multiple replicas of an MPI process. Thus, while VolpexMPI can continue the execution of an application seamlessly in case of a process failure if replicas are available for that process, MPICH-V and RADICMPI will have to deal with the overhead generated by restarting processes from an earlier checkpoint. Also, message logging in VolpexMPI is fully distributed on host nodes themselves and there is no equivalent of channel memories.

MPI/FT [16] provides transparent fault tolerance by replicating MPI processes and introducing a central coordinator. The library is able to recognize malicious data by using a global voting algorithm among replicas. However, this feature also leads to an exponential increase in the number of messages with the number of replicas: each replica sends every message to all destination replicas. VolpexMPI avoids the penalty resulting from this communication scheme by ensuring that a message is pulled from exactly one sender with receiver initiated communication. P2P-MPI [17] is also based on replication techniques where each set of process replicas maintain a master replica that distributes messages. Fault detection is done using a gossip-style protocol [18], which has the ability to scale well and provides timely detection of failures. P2P-MPI also takes advantage of locality awareness and co-allocation strategies. A key difference is that, unlike P2P-MPI, VolpexMPI utilizes a pull based model for data communication that ensures that the application advances at the speed of the fastest replica for each process.

VolpexMPI also employs replication for fault tolerance. It has been shown, that check-pointing offers a good solution when failures are infrequent whereas replication offers better performance when failure rates are high [19]. VolpexMPI is designed to balance replication and checkpoint-restart, although the current implementation is limited to replication.

3 VolpexMPI Design

VolpexMPI is an MPI library implemented from scratch focusing on fault tolerance using process replication. As of today, the library supports around 40 functions of the MPI-1 specification. The design of the library is centered around five major building blocks, namely the MPI API layer, the point-to-point communication module, a buffer management module, a replica selection module and a data transfer module.

The point-to-point communication module of VolpexMPI has to be designed for MPI processes with multiple replicas in the system. This is required in order to handle the main challenge of grids built from idle PCs, namely the fact that processes are considered fundamentally unreliable. A process might go away for no obvious reason, such as the owner pressing a button on the keyboard. From the communication perspective, the library has two main goals: (I) avoid increasing the number of messages on the fly by a factor of $n_{replicas} \times n_{processes}$, i.e., every process sending each message to every replica, and (II) make the progress of the application correspond to the fastest replica for each process.

In order to meet the first goal, the communication model of VolpexMPI deploys a receiver initiated message exchange between processes where data is pulled by the receiver from the sender. In this model, the sending node only buffers the content of a message locally, along with the message envelope. Furthermore, it posts for every replica of the corresponding receiver rank, a non-blocking, non-expiring receive operation. When contacted by a receiver process about a message, a sender participates in the transfer of the message if it is buffered, otherwise informs the receiver that the message is not available.

The receiving process polls a potential sender and waits then for the data item or a notification that the data is not available. As of today, VolpexMPI does not support wildcard receive operations as an efficient implementation poses a significant challenge. A straight-forward implementation of `MPI_ANY_SOURCE` receive operations is possible, but the performance would be significantly degraded compared to non-wildcard receive operations.

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 buffer management module 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. Our current approach employs a circular buffer where the oldest log entry is removed when the buffer is full. The long-term goal is to coordinate the size of the circular buffer with checkpoints of individual processes, which will allow guaranteed restarts with a bounded buffer size.

In order to meet the goal that the progress of an application correspond to the fastest replica for each process, the library has to provide an algorithm which allows a process to generate an order in which to contact the sender replicas. This is the main functionality provided by the replica-selection module. The algorithm utilized by the replica-selection module has to handle two seemingly contradicting goals: on one hand, it would be beneficial to contact the "fastest" replica from the performance perspective. On the other hand, the library does not want to slow-down the fastest replica by making it handle significantly larger number of messages, especially when a message is available from another replica. The specific goal, therefore, is to determine a replica which is "close" to the execution state of the receiver process. Currently the library utilizes a simple approach which groups replicas into 'teams'. A receiver tries to contact the first the replica within its team, and only contacts a replica of another team if its own replica does not response within a given time slot. This is, however, a topic of active research with more sophisticated algorithms in the process of being implemented.

The data transfer module of VolpexMPI relies on a socket library utilizing non-blocking sockets. In the context of VolpexMPI, the relevant characteristics of this socket library are the ability to handle failed processes, on-demand connection setup in order to minimize the number of network connections, an event delivery system integrated into the regular progress engine of the socket library and the notion of timeouts for both communication operations and connection establishment. The latter feature will be used in future versions of VolpexMPI to identify replicas which are lagging significant.

The startup of a VolpexMPI application utilizes a customized `mpirun` program which takes the desired replication level as a parameter, in addition to the number of MPI processes, the name of the executable, and a list of hosts where the processes shall be started. As of today, the startup mechanism relies on secure shell operations. However, we anticipate to extend this section of the code in the near future by customized BOINC or CONDOR functions. `mpirun` has furthermore the functionality to inform all MPI processes about their rank, the team they belong to, as well as the information required by an MPI process to contact any other process within this application.

4 Experiments and Results

This section describes the experiments with the VolpexMPI library and the results obtained. VolpexMPI has been deployed on a small cluster as well as pool of desktop PCs. Although VolpexMPI is designed for PC grids and volunteer environments, experimental results are shown for a regular cluster in order to determine the fundamental performance characteristics of VolpexMPI in a stable and reproducible environment. The 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. For evaluation we utilize the Gigabit Ethernet network interconnect of the cluster to compare VolpexMPI run times to Open MPI [20] v1.2.6. and examine the impact of replication and failure on performance.

First, we document the impact of the VolpexMPI design on the latency and the bandwidth of communication operations. For this, we ran a simple ping-pong benchmark using both Open MPI and VolpexMPI on the cluster. The results shown in Figure 1 indicate, that the receiver based communication scheme used by VolpexMPI can achieve close to 80% of the bandwidth achieved by Open MPI. The latency for a 4 byte message increases from roughly 0.5ms with Open MPI to 1.8ms with VolpexMPI. This is not surprising as receiver based communication requires a ping-pong exchange before the actual message exchange.

Next, the NAS Parallel Benchmarks (NPBs) are executed for various process counts and data class set sizes. 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. Since VolpexMPI targets the execution of applications with moderate number of processes, we present results obtained for 8 process and 16 process scenarios.

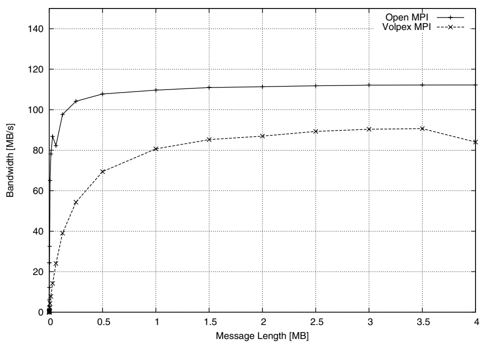
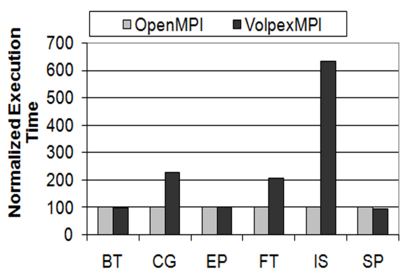


Fig. 1. Bandwidth comparison of OpenMPI and VolpexMPI using a Ping-Pong Benchmark



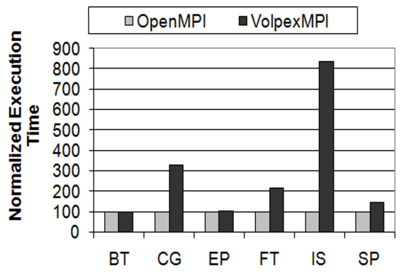


Fig. 2. Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 Processes (left) and 16 Processes (right)

Figure 2 shows results for runs of 8 processes (left) and 16 processes (right) utilizing the Class B data sets for six of the NPBs. We have excluded LU and MG from our experiments due to their use of `MPI_ANY_SOURCE` which is not currently supported in VolpexMPI. These reference executions did not employ redundancy (x1). The run times for Open MPI are shown for comparison in the bar graph. All times are noted as normalized execution times with a reference time of 100 for Open MPI. The overhead incurred in the VolpexMPI implementation is virtually non-existent for BT, SP, and EP for the 8 and 16 processes, except that SP shows a noticeable overhead of 45% for 16 processes. The overhead for CG, FT and IS is significantly higher due to a variety of reasons, such as a greater use of collective calls such as `MPI_Alltoall(v)`, and in the case of IS, a ratio of computation to communication which is unfavorable to higher-latency environments. This also documents the fact that the class of applications considered suitable for execution with VolpexMPI have to follow a sparse communication scheme, i.e., a process should optimally only communicate with a small number of other processes, and should have a favorable communication to computation ratio. These requirements broadly hold for BT, SP and EP, but not necessarily for CG, FT and IS.

Next, we document the effect of executing an application with multiple copies of each MPI process. The left part of Figure 3 shows the normalized execution times of VolpexMPI for the 8 process NPBs 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 VolpexMPI scheme/model. Note, that this is a significant improvement over the replication based related work in the field. The benchmarks that show some sensitivity to replication are CG and SP, and the reasons are currently under investigation. Since Open MPI is not designed for utilizing redundant nodes, there are no directly comparable results for the double redundancy (x2) and triple redundancy (x3) runs.

Finally, we document the performance impact of a process failure for the NAS Parallel Benchmarks when using VolpexMPI. 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

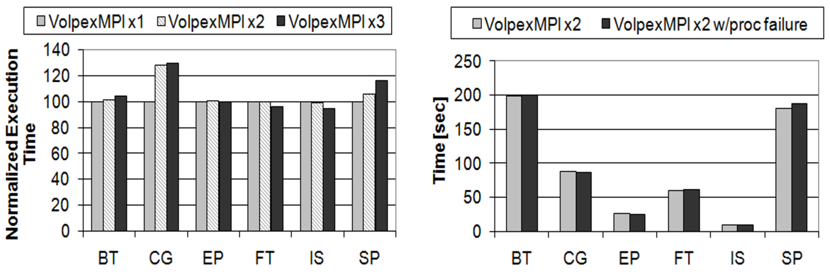


Fig. 3. Comparison of VolpexMPI execution times for 8 MPI processes with varying degree of replication (left) and in case of a process failure (right)

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 VolpexMPI, 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 the right part of Figure 3 show virtually no overhead in the scenario outlined above compared to the fault-free execution of the same benchmark using double redundancy. There are two potential sources for overhead. The first comes from the fact that the surviving process with rank 1 is being queried for data items by the second "team" of processes. Second, detecting the process failure is as of today based on a timeout mechanism or the break-down of a TCP connection. However, since the correct result of the simulation is available as soon as any replica of each process finishes the execution, these overheads might not necessarily show up in the final result.

5 Conclusions

This paper introduces VolpexMPI, an MPI library designed for robust execution of parallel applications on PC grids. The key design goal is efficient execution with replicated processes. The library employs a receiver based communication model between the processes, and a distributed, sender based message logging scheme.

We demonstrated with a prototype implementation the necessity to focus on the right class of applications for Volpex MPI, namely those with a favorable communication to computation ratio and a modest degree of communication. Benchmarks having the required characteristics show only a minor overhead compared to a standard MPI library. More important, utilizing multiple replicas for each process does not impose a notable overhead for the majority of the NAS benchmarks. Also, the NAS Parallel Benchmarks analyzed could successfully survive a process failure without suffering a major performance degradation. Hence, the central functionality and performance goals for VolpexMPI are satisfied.

The ongoing work on VolpexMPI includes developments in algorithms and execution environment. We are working on integrating checkpoint-restart with Volpex-MPI to dynamically manage replication by recreating slow and failed replicas from healthy replicas. We also plan to deploy and evaluate VolpexMPI on a large campus PC grid. Currently CONDOR and BOINC are being investigated as vehicles for integrated deployment.

We are investigating several applications as candidates for execution on PC clusters and building simulation tools to rapidly assess the suitability of an application for Volpex-MPI. In particular, we are in active discussions with a research group at the University of Houston which develops the Replica Exchange Molecular Dynamics (REMD) application [9]. The memory and compute requirements of this application combined with its low but important communication requirements make the application ideally suited for VolpexMPI.

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. Anderson, D., Fedak, G.: The computation and storage potential of volunteer computing. In: Sixth IEEE International Symposium on Cluster Computing and the Grid (May 2006)
2. Kondo, D., Taufer, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating desktop grids: An empirical study. In: International Parallel and Distributed Processing Symposium, IPDPS 2004 (April 2004)
3. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency - Practice and Experience 17(2-4), 323–356 (2005)
4. Anderson, D.: Boinc: A system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing (November 2004)
5. Amazon webservices: Amazon Elastic Compute Cloud, Amazon EC2 (2008), `http://www.amazon.com/gp/browse.html?node=201590011`
6. Google Press Center: Google and IBM Announce University Initiative to Address Internet-Scale Computing Challenges (October 2007), `http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html`
7. Tabe, T., Stout, Q.: The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan (November 1999)
8. Kerbyson, D., Barker, K.: Automatic identification of application communication patterns via templates. In: Proc. 18th International Conference on Parallel and Distributed Computing Systems (PDCS 2005), Las Vegas, NV (September 2005)
9. Sugita, Y., Okamoto, Y.: Replica-exchange molecular dynamics method for protein folding. Chemical Physics Letters 314, 141–151 (1999)
10. Case, D., Pearlman, D., Caldwell, J.W., Cheatham, T., Ross, W., Simmerling, C., Darden, T., Merz, K., Stanton, R., Cheng, A.: Amber 6 Manual (1999)

11. Kanna, N., Subhlok, J., Gabriel, E., Cheung, M., Anderson, D.: Redundancy tolerant communication on volatile nodes. Technical Report UH-CS-08-17, University of Houston (December 2008)
12. Fagg, G.E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J.J.: Process fault-tolerance: Semantics, design and applications for high performance computing. International Journal of High Performance Computing Applications 19, 465–477 (2005)
13. Ltaief, H., Gabriel, E., Garbey, M.: Fault Tolerant Algorithms for Heat Transfer Problems. Journal of Parallel and Distributed Computing 68(5), 663–677 (2008)
14. Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., Magniette, F.: Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In: SC 2003: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, vol. 25. IEEE Computer Society, Los Alamitos (2003)
15. Duarte, A., Rexachs, D., Luque, E.: An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 150–157. Springer, Heidelberg (2006)
16. Batchu, R., Neelamegam, J.P., Cui, Z., Beddhu, M., Skjellum, A., Yoginder, D.: Mpi/ft tm: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In: Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid, pp. 26–33 (2001)
17. Genaud, S., Rattanapoka, C.: Large-scale experiment of co-allocation strategies for peer-to-peer supercomputing in p2p-mpi. In: IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, pp. 1–8 (2008)
18. Van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. Technical report, Ithaca, NY, USA (1998)
19. Zheng, R., Subhlok, J.: A quantitative comparison of checkpoint with restart and replication in volatile environments. Technical Report UH-CS-08-06, University of Houston (June 2008)
20. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B.W., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)