Influence of the Progress Engine on the Performance of Asynchronous Communication Libraries*

Edgar Gabriel *

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-10-02

May 3, 2010

Keywords: volunteer computing, cluster computing, performance analysis, message passing libraries

Abstract

This technical report performs an in-depth performance comparison of two MPI libraries, namely VolpexMPI and Open MPI. The analysis is motivated by some unexpected results in which VolpexMPI shows better performance than Open MPI, despite of some architectural decision in the library that should lead to a performance degradation on a dedicated compute cluster. Our analysis indicate that general purpose high performance computing communication libraries are optimized for high speed network interconnects such as InfiniBand, which due to their low latency and high bandwidth require an aggressive approach in pushing data into the network. This approach is however not necessarily optimal for a Gigabit Ethernet network. Specifically, the progress function of the communication library is called more often than necessary to saturate the Gigabit Ethernet network, which consequently introduces an overhead.

*. 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.

*Department of Computer Science, University of Houston.

Influence of the Progress Engine on the Performance of Asynchronous Communication Libraries*

Edgar Gabriel *

Abstract

This technical report performs an in-depth performance comparison of two MPI libraries, namely VolpexMPI and Open MPI. The analysis is motivated by some unexpected results in which VolpexMPI shows better performance than Open MPI, despite of some architectural decision in the library that should lead to a performance degradation on a dedicated compute cluster. Our analysis indicate that general purpose high performance computing communication libraries are optimized for high speed network interconnects such as InfiniBand, which due to their low latency and high bandwidth require an aggressive approach in pushing data into the network. This approach is however not necessarily optimal for a Gigabit Ethernet network. Specifically, the progress function of the communication library is called more often than necessary to saturate the Gigabit Ethernet network, which consequently introduces an overhead.

Index Terms

volunteer computing, cluster computing, performance analysis, message passing libraries

I. 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.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.

Recently, we have introduced VolPEx (Parallel Execution on Volatile Nodes) MPI [7], a comprehensive and scalable solution to execute parallel scientific applications on virtual clusters composed of volatile ordinary PC nodes. In order to cope with the characteristics such as frequent node failures and strongly varying compute and communication characteristics, Volpex MPI deploys two (or more) replicas for each process and uses a sender based message logging in order to deliver messages to lagging processes due to slow execution or recreated processes from a checkpoint.

The goal of this technical report is to give details on the performance analysis of VolpexMPI and Open MPI, since for a number of test cases we observed results that are neither intuitive nor easy to explain. The remainder

*. 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.

*Department of Computer Science, University of Houston.

of the technical report is organized as follows: section II presents briefly the design of Volpex MPI. At the core of the paper is the performance analysis of VolpexMPI in section III along with detailed explanations on the reasons for the performance observed. Finally, section IV summarizes the findings and presents the ongoing work on the library.

II. Design and Implementation of VolpexMPI

VolpexMPI is an MPI library targeting volunteer compute environment. The most relevant characteristics of volunteer environments are a) the fundamental unreliability of compute processes that might go away for virtually no reason (e.g. pressing a key on the keyboard by the owner), and b) the distributed nature of the compute environment. The key features of VolpexMPI therefore are:

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, although this aspect is not yet available.
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.

Internally, 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.

In the following, we focus on the design and implementation of the point-to-point operations, since this yields the most significance for the subsequent performance analysis.

A. Point-to-Point Communication

The point-to-point communication module of VolpexMPI has to be designed for MPI processes with multiple replicas in the system. The library has two main goals for its point-to-point communication: (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 replies with the requested item as soon as it is available.

The receiving process polls a potential sender and waits then for the data item to be available. A timeout mechanism can be used to handle the situation where a receiver process requests a data item that is not in the local buffer of the sender process anymore. 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.

B. Data Transfer

The data transfer module of VolpexMPI relies on a socket library utilizing non-blocking sockets. Non-blocking sockets allow for a gradual sending/receiving of data items due to the fact that a `write()` or `read()` operation

will immediately return if the current status of the communication channels do not allow for progress at this point in time. This allows to efficiently manage a large number of network connections without having to deal with multiple threads, and avoids deadlock situations that might occur when using standard, blocking sockets. A key component of the library is its progress function, that checks in each invocation which of the currently registered socket descriptors are ready for the next data transfer operation. For this, the progress function utilizes internally the `select()` operation, and maintains for each process the next element to be sent/received, and a pointer to the position where the last transfer has stopped. Similarly to most standard MPI libraries, VolpexMPI invokes its progress function at least once in every MPI function.

Further relevant characteristics of the 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 and the notion of timeouts for both communication operations and connection establishment. The latter feature is used to identify replicas which are lagging significant. A communication operation which exceeds the pre-defined timeout of the library might not necessarily be dead but may correspond to a significantly slower replica than others, and hence it might be appropriate to abort the communication operation. Thus, using the target selection functionality, process will re-direct their communication requests to other existing replicas of the according MPI process.

III. Experiments and results

This section describes the experiments with the VolpexMPI library and the results obtained on a dedicated cluster. Although VolpexMPI is designed for PC grids and volunteer environments, the experiments shown on the dedicated cluster are performed in order to determine the fundamental performance characteristics of VolpexMPI as compared to the results in a stable and reproducible environment.

The dedicated cluster utilizes 29 compute nodes with 88 cores total, 24 nodes 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 [8] v1.4.1. 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 dedicated cluster. The results 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.

Figure 1 shows results for runs of 8 processes (upper left), 16 processes (upper right), 32 processes (lower left) and 64 processes (lower right) for 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 currently not 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, and are the average of three runs.

The results indicate, that in the majority of the test cases VolpexMPI performs as well or even better than Open MPI using the TCP interfaces. While this is acceptable for the EP benchmark since it does not contain any (significant) communication, it is surprising for the communication intensive benchmarks such as FT, due to the documented overhead of VolpexMPI compared to Open MPI. In the following, we would like to confirm a) the correctness of the results delivered by the VolpexMPI executions and b) explain the performance.

In order to confirm the correctness of the results, we went through a multi-step procedure. First, all NAS Parallel Benchmarks have a verification step built in. All benchmarks executed using VolpexMPI succeeded the NPB built-in verification. Second, using a small library which calculates a CRC-32bit checksum of a message we confirmed, that the number of messages, sequence of messages and the content of every single message is absolutely identical for

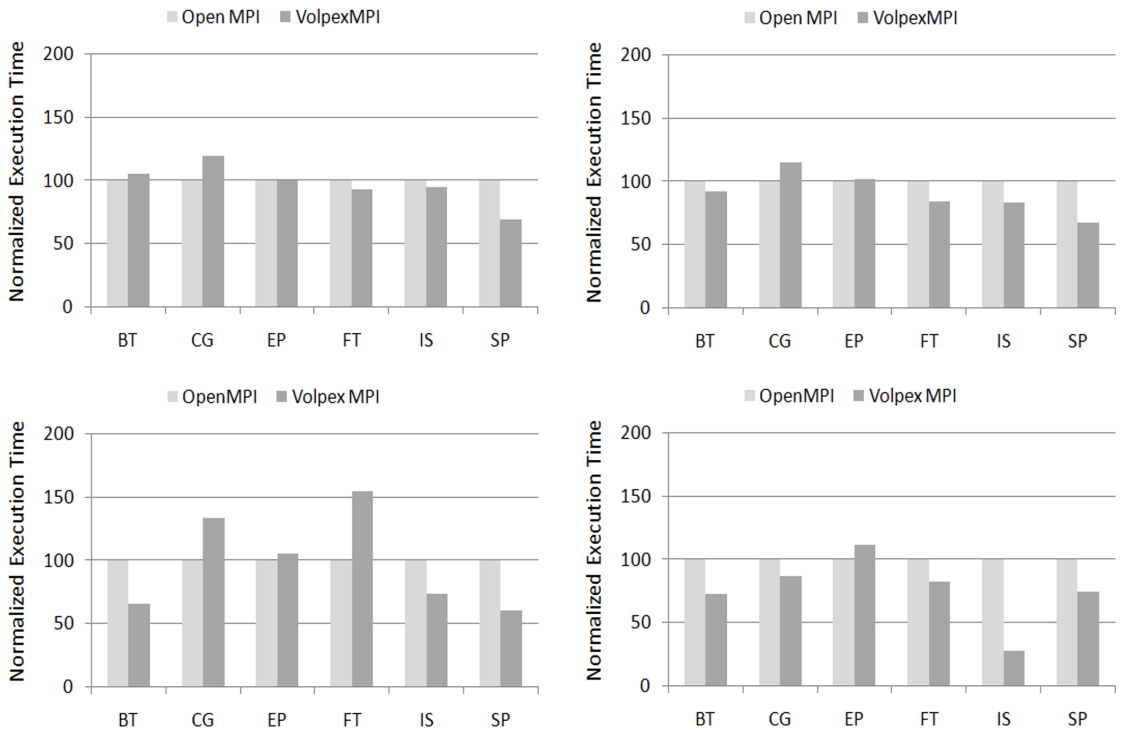

Fig. 1. Comparison of Open MPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 processes (upper left), 16 processes (upper right), 32 processes (lower left) and 64 processes (lower right) on a dedicated cluster.

the VolpexMPI and the Open MPI executions on every single process. Finally, in order to exclude the possibility, that we hit a performance problem within the Open MPI library, selected test cases have also been executed MPICH2 version 1.0.7. The performance results obtained this MPI library were generally in the same range as the Open MPI performance numbers with minor deviations in both directions.

In the following, we detail the reasons for the performance differences between VolpexMPI and a standard MPI library such as Open MPI. To summarize the findings of the section, libraries such as Open MPI are optimized for high speed network interconnects such as InfiniBand, which due to their low latency and high bandwidth require an aggressive approach in pushing data into the network by calling their progress function whenever possible, e.g. in each MPI function. This approach is however not optimal for a Gigabit Ethernet network. Specifically, the progress function is called more often than necessary to progress data on this network, which consequently introduces an overhead especially for send-sockets due to the required matching of sockets to processes and the according look-up operation whether data is ready for being sent over the sockets.

The receiver based communication scheme of VolpexMPI introduces a slight overhead per message, that results in a delay in between subsequent calls to the (lower level) progress engine. Since the progress engine is only called from MPI functions, the time between calling the VolpexMPI (low level) progress function is determined by two components: the time spent outside of MPI functions, i.e. time spent in computation, and the delay introduced by the VolpexMPI receiver based communication scheme. There are three scenarios to be distinguished.

First, if the time spent in computation is significantly larger than the time spent communication, the overhead of the VolpexMPI communication schemes will not have a major influence on the overall execution time. This is mostly the case for the 8 and 16 processes testcases shown above. Some of the VolpexMPI numbers are lower than the OpenMPI numbers even for these testcases, which we attribute to two facts: processes in VolpexMPI are more 'decoupled' compared to a regular MPI library, since a send operation only copies data into a local buffer and never blocks. This is especially evident e.g. for SP, since it allows for a more efficient overlapping of communication and computation. Further, as detailed in subsection III-A, the algorithms used for collective operations in VolpexMPI perform better over this Gigabit Ethernet network than the default algorithms used in Open MPI.

Second, if the time spent in computation is not dominant on a per process basis, the additional delay introduced

by VolpexMPI will result in a performance penalty due to the fact that the progress engine is not called often enough to saturate the network. This is e.g. the case for the 32 process CG and FT test cases.

Third, if the time spent in computation is small, as is the case e.g. for the 64 process test cases shown above, the additional delay helps to reduce the number of times the progress engine is called compared to Open MPI, while its still being called often enough to saturate the network. Subsection **??** emphasizes this point by providing precise numbers and measurements.

A. A Case Study with Detailed Analysis

In this section, we elaborate in more details on the performance gain of VolpexMPI compared to Open MPI. For the sake of simplicity, we choose for the subsequent analysis a single testcase, namely 64 processes testcase of IS, since it showes the most glaring difference in the execution time. Table I details the overall execution time spent in different MPI routines by the benchmark for both Open MPI and VolpexMPI, based on data achieved using a profiling library. The main result of this analysis is, that this benchmark is entirely dominated by communication operations, and two functions in particular: `MPI_Allreduce` and `MPI_Alltoallv`. These functions also show the most significant difference in the execution time of between VolpexMPI and Open MPI. Note, that the sum of the execution time shown in the table is larger than the overall execution time indicated by the NPB benchmarks, which is due to the fact that we show the maximum execution time for each function across all processes, and the tracing library also includes data from communication operations that occur outside of the timing loop in the NAS benchmark. The profiling library further revealed, that the typical message length of the Allreduce operation was 8 bytes or 1 double precision number, and for the Alltoallv operation it was in between 27k and 37k for each pair of processes. To simplify the analysis of the Alltoallv operation, we assume for the subsequent paragraphs an average message length of 32k between each pair of processes.

	Open MPI	VolpexMPI
overall time	28.46 sec	7.12
`MPI_Reduce`	1.6 sec	0.29 sec
`MPI_Allreduce`	19.7 sec	4.11 sec
`MPI_Alltoall`	0.25 sec	1.8 sec
`MPI_Alltoallv`	21.5 sec	3.2 sec
`MPI_Irecv`	0.000005 sec	0.0001 sec
`MPI_Send`	0.0001 sec	0.0001 sec
`MPI_Wait`	2.43 sec	0.328 sec

TABLE I

BREAKDOWN OF THE EXECUTION OF THE 64 PROCESSES IS TEST CASE FOR OPEN MPI AND VOLPEXMPI.

The first question arising with respect to the performance of the Allreduce and Alltoallv operations is how the implementations of these operations differ between Open MPI and VolpexMPI. VolpexMPI uses linear algorithms without any major optimizations. On the other hand, the default Open MPI collective module (*tuned*) has a collection of state-of-the-art algorithms for both operations. However, it turns out, that these algorithms are designed and optimized for low-latency networks such as InfiniBand, and deliver suboptimal performance on a commodity network such as Gigabit Ethernet. By switching to the *basic* module for collective operations, which uses mostly linear algorithms similar to VolpexMPI, the time spent in Allreduce operations has been reduced from 19 seconds to around 9 seconds, and the time for Alltoallv operations decreased from 21 seconds to 10.5 seconds. Thus, the performance of IS using the *basic* collective module has been reduced to roughly 15 seconds, which is still more than a factor two above the VolpexMPI execution time for the very same test.

In order to detail where the performance difference comes from, we focus now entirely on the Alltoallv operation, assuming that each pair of processes exchange 32kb of data. We implemented the Alltoallv operation in a simply test code as a simple sequence of non-blocking send and receive operations followed by a waitall function as outlined below:

```
MPI_Comm_size ( comm, &size );
for ( i=0; i<size; i++ ) {
```

```
   MPI_Isend (sbuf[i], scnt[i], MPI_BYTE, i, 0, comm, &reqs[2*i]);
   MPI_Irecv (rbuf[i], rcnt[i], MPI_BYTE, i, 0, comm, &reqs[2*i+1]);
}
MPI_Waitall ( 2*size, reqs, stats);
```

Our benchmark provides two different version of this code: the first version relies on the VolpexMPI semantics that uses sender-side buffering and a receiver-based message retrieval scheme. The second version uses a direct communication scheme similarly to Open MPI, by using the low-level data transfer operations of the socket library used underneath the hood by VolpexMPI. The execution time of the second code version for one Alltoallv operation for 64 processes on the Gigabit Ethernet network is around 1.5 seconds, similarly to the execution time of the very same code sequence over Open MPI. Version 1 of the code using the VolpexMPI scheme takes however only 0.9 seconds. In both cases, the entire time is spent in the progress engine of the low-level socket library. Therefore, we instrumented the progress function of the socket library to retrieve some more details on the behavior. Table II shows the most important number of this analysis when executing 10 Alltoallv operations for both versions.

	VolpexMPI direct communication	VolpexMPI receiver based commmunication
# called	690,545	11,981
# recv sockets kicked off	25,172	6,112
# send sockets kicked off	44,194,880	766,784
time handling recv sockets	340896.00 usec	212753.00 usec
time handling send sockets	8439059.00 usec	428231.00 usec

TABLE II

DETAILED ANALYSIS OF THE PROGRESS ENGINE USAGE FOR THE DIRECT COMMUNICATION SCHEME AND THE RECEIVER BASED COMMUNICATION SCHEME.

The first notable result is, that for the direct communication scheme the progress engine is called more than 57 times more often than in the receiver based communication scheme. Since the progress engine has to handle both send and receive sockets separately when using non-blocking sockets, we also detail how often each of these two types of sockets have reported to be ready for the next transaction by the `select` call in the progress engine, and how much time is spent in dealing with both send and receive sockets. The results indicate, that the receiver side sockets are behaving very similarly for both communication schemes. Although four times as many receive sockets have kicked off for the direct communication scheme as for the receiver based communication scheme, the difference in the time spent for handling receive sockets is less than a factor of two compared to the receiver based communication scheme. Especially when putting it into relation to the fact, that the progress is called 57 times as often for the according scenario, this factor of 1.6 is nearly negligible.

The situation is however different for send sockets. When using non-blocking sockets, the select function indicates that data can be sent over that connection every time the resource is not busy. The progress engine of VolpexMPI (as well as of Open MPI and other MPI libraries) has to check in that case what process that particular TCP socket is associated to, and whether there is any data item currently enlisted as 'ready to send' to that process. Due to the fact, that the progress engine is called 57 times more often for this scenario when using the direct communication scheme, we have in fact 57 times more send sockets indicating that they are ready for sending data, which means we have 57 times more operations to look up to which process this socket belongs to and whether there is data to be sent. Not surprisingly, we spend more than 20 times more time in handling send-sockets for the direct communication scheme, compared to the receiver based communication scheme used by VolpexMPI.

To summarize the findings, the performance benefit of VolpexMPI compared to Open MPI for many of the testcases shown above comes from the fact, that the receiver based communication scheme introduces a (small) overhead in processing messages, that reduce the aggressivenes with which the progress engine of the lower-level socket library is being called. As long as the frequency of calling the progress engine is high enough to ensure saturation of the network, this less aggressive strategy helps to reduce the overall costs of communication due to the fact, that non-blocking sockets are virtually always ready for sending data, and the associated lookup operations are not entirely for free if executed often enough.

IV. Summary and Concluding remarks

To summarize the findings of the analysis, libraries such as Open MPI are optimized for high speed network interconnects such as InfiniBand, which due to their low latency and high bandwidth require an aggressive approach in pushing data into the network by calling their progress function whenever possible. This approach is however not necessarily optimal for a Gigabit Ethernet network. Specifically, the progress function is called more often than necessary to saturate the Gigabit Ethernet network, which consequently introduces an overhead.

However, since the overall delay between two subsequent calls to the progress engine is also dependent on the amount of time spent in computation between two subsequent calls to an MPI function, results with other applications might in fact deviate from the benchmark results shown here. Our overall findings in this technical paper are therefore, that the overhead introduced by the VolpexMPI communication scheme is all-in-all low for applications having favorable communication characteristics, and we would not expect the results to be easily transferable or representative for other applications and platforms.

References

[1] D. Anderson and G. Fedak, "The computation and storage potential of volunteer computing," in *Sixth IEEE International Symposium on Cluster Computing and the Grid*, May 2006.

[2] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien, "Characterizing and evaluating desktop grids: An empirical study," in *International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004. [Online]. Available: citeseer.ist.psu.edu/kondo04characterizing.html

[3] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[4] D. Anderson, "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)," http://www.amazon.com/gp/browse.html?node=201590011, 2008.

[6] Google Press Center, "Google and IBM Announce University Initiative to Address Internet-Scale Computing Challenges," http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html, October 2007.

[7] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok, "VolpexMPI: an MPI Library for Execution of Parallel Applications on Volatile Nodes," in *Proc. The 16th EuroPVM/MPI 2009 Conference*, Espoo, Finland, 2009, pp. 124–134, lecture Notes in Computer Science, volume 5759.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.