A Robust Communication Framework for Parallel Execution on Volunteer PC Grids

Eshwar Rohit*, Hien Nguyen*, Nagarajan Kanna*, Jaspal Subhlok*, Edgar Gabriel*,
Qian Wang†, Margaret S. Cheung†, David Anderson‡

* Department of Computer Science, University of Houston
† Department of Physics, University of Houston
‡ UC Berkeley Space Sciences Laboratory

***Abstract*—Volunteer PC grids represent massive computation capacity at a low cost, but are challenging to employ for parallel computing because of variable and unpredictable performance and availability. A communicating parallel program must employ explicit redundancy, or implicit redundancy with uncoordinated checkpoint-restart to make continuous forward progress in such an unreliable environment. A communication model based on one-sided Put/Get calls to an abstract global shared space is a good match as processes can execute their communication operations independently and asynchronously. However, no existing system is designed for redundant communicating processes. The key problem is that a single logical operation that impacts the global program state may be executed by different instances of the same process at different times leading to semantic inconsistency. This paper presents the design, execution model, implementation, and usage of *Volpex*, a communication layer for robust execution on volunteer PC grids. The research leads to a practical way to employ idle PCs for latency tolerant parallel computing applications.**

I. Introduction

In recent years ordinary desktops and PCs have been employed successfully for large scale scientific computing, most commonly using Condor [1] or BOINC [2] as middleware. The Condor scheduler enables ordinary desktops to be employed for compute intensive applications. It is deployed at over 850 known sites with at least 125,000 hosts around the world. The BOINC middleware uses volunteered public PCs for scientific applications when idle. It has been remarkably successful, managing over half a million nodes and over 30 scientific research projects since its release in 2004. However, the target applications for BOINC and CONDOR are generally limited to master-slave or bag-of-tasks parallelism.

We will refer to PCs made available for scientific computing when idle as *volunteer nodes or PCs* and an execution environment composed of volunteer nodes connected by a LAN or Internet as a volunteer computing environment (even if the PCs are in an organization with no actual volunteering). Volunteer nodes represent a potentially immense but *volatile* resource, i.e., they are heterogeneous and their availability to execute guest scientific applications can change suddenly and frequently based on the desktop owner's actions. Execution of communicating parallel applications on volunteer nodes is extremely challenging because process failures and slowdowns are frequent and the failure or slowdown of a single process impacts the entire application. Hence, a mechanism for fault tolerance is practically a requirement. If a checkpoint-restart approach is used, the checkpoints must be taken independently and asynchronously, because of the potential overhead of global synchronization. For this approach, the communication framework must be able to respond to communication requests during recovery, which may be duplicate requests corresponding to a communication operation that was executed in a past state of program execution. Similar functionality is required when redundancy is employed for fault tolerance, an approach that becomes more attractive in high failure scenarios [3]. With redundancy, multiple physical processes in different states co-exist for a single logical process, and the communication framework must be able to respond to redundant communication requests from process replicas.

Implementation of MPI style message passing, the dominant paradigm for parallel programming today, is problematic in such scenarios because of the synchronous nature of message transfers. Put/Get style asynchronous communication pioneered by Linda [4] is potentially a good fit for communication on volunteer nodes as it provides an abstract global shared space that processes can use for information exchange without a temporal or spatial coupling. However, redundant processes or asynchronous recovery from checkpoints, is not supported in existing systems that provide an abstract global shared space.

This paper introduces the Volpex dataspace API for anonymous Put/Get style communication among processes. The API and its execution model can support common message passing and shared memory programming styles. The key additional requirement is correct and efficient support for multiple physical Put/Get requests corresponding to a single logical Put/Get request. All requests corresponding to a unique logical request are satisfied in the same manner with identical data objects. This allows support of implicit (due to checkpoint-restart) or explicit redundant execution with the final results guaranteed to be identical to (one of the possible) results with normal execution. Management of execution is orthogonal to this process; the communication infrastructure as well as application processes are unaware whether, and to what extent, redundancy is being employed, or if it is implicit or explicit. The core requirement is to be able to handle

asynchronous process replicas in any state of execution.

This dataspace communication API is a component of the Volpex framework (Parallel Execution on Volunteer nodes) that attempts to achieve seamless forward application progress in the presence of routine failures by employing redundancy and checkpointing. The primary goal is to transform ordinary PCs into virtual clusters to run a variety of parallel codes. However, the methods developed in the paper are potentially applicable to other scenarios also, such as employing unused process cores to run replicas to improve reliability. The Volpex dataspace API is designed for applications with low to moderate communication requirements. It is expected to scale to 100s of nodes on institutional LANs. While many applications are excellent candidates for Volpex, certainly not all applications will run effectively on ordinary desktops under Volpex (or any other framework) because of memory and communication requirements that can only be met with dedicated clusters.

The paper presents the design, execution model, implementation, and usage results for the Volpex dataspace API. Codes developed span different types of usage of dataspace API and include an actively used *Replica Exchange Molecular Dynamics* application. The dataspace API communication is integrated with BOINC framework for a comprehensive solution to parallel computing on volunteer nodes.

II. Related work

Idle desktops are widely used for parallel and distributed computing. The Berkeley Open Infrastructure for Network Computing (BOINC) [2] is a middleware system widely used for volunteer computing where people donate the use of their computers to help scientific research. Condor [1], is a workload management system that can effectively harness wasted CPU power from otherwise idle desktop workstations. Other systems that build desktop computing grids include Entropia [5], iShare [6], and OurGrid [7]. Mechanisms applied for fault tolerance in PC grids, such as redundancy in BOINC and checkpointing in Condor [8] are important for long running sequential and bag-of-task codes, but are generally not sufficient for communicating parallel programs.

Linda [4] has been an active research topic for over two decades. It represents a model of coordination and communication among parallel processes based on logically global associative memory, called a *tuplespace*. There are a number of variants of Linda available, such as TSpaces [9], JavaSpaces [10], and SALSA [11], a Linda adaptation for molecular dynamics applications.

There has been considerable work in fault tolerance in Linda, but it has largely focused on making the Linda tuplespace itself resilient to failure. A replication based fault tolerant implementation of Linda tuplespace is discussed in [12]. FT-Linda [13] provides a stable tuple space that persists across failures and atomic tuple space transactions that allow development of some types of fault tolerant applications. PLinda [14] provides transactional mechanisms to achieve atomic operations and process-private logging that processes can utilize for checkpoint-restart mechanisms. We have employed some of the ideas, in particular, atomic operations. However, none of these (and other) frameworks provide transparent processing of arbitrary replicated communication requests. The notion of having a distributed shared memory programming model using an abstract data space has also been explored in [15], without however, support for fault tolerance. This paper reports on development of fault tolerant parallel execution with an abstract shared memory with support for replicated processes and processes restarted from local checkpoints.

Several implementations of the MPI specification have focused on deploying fault-tolerance mechanisms. The vast majority of projects in that field rely on system level checkpointing and automatic roll back of the application, often relying on a third party system level library such as BLCR [16]. Representatives of these libraries include StarFish MPI [17], Egida [18], Open MPI [19], MPICH-V [20], and Adaptive MPI [21].

A smaller number of approaches have utilized process replication to create robust MPI libraries, such as MPI/FT [22], P2P-MPI [23], rMPI [24] and VolpexMPI [25], the latter being developed as a complementary aspect of the work presented in this paper. P2P-MPI and VolpexMPI are the only libraries targeting volunteer computing within this group, with P2P-MPI supporting only Java based applications, which is a major restriction for many scientific applications.

Process replication is also deployed in the MOON framework [26], which represents a robust model for MapReduce applications for volunteer computing. MOON deploys a small number of dedicated servers along with volunteer resources. However, this solution is for applications that fit the MapReduce model, in contrast to a general communication API and execution environment developed by Volpex.

III. Dataspace programming model

The programming model developed consists of independent processes communicating through an abstract *dataspace*. The key objective was that the execution model allow multiple and varying number of instances of each process with the application execution state advancing with the fastest replica for each process. The design was driven by simplicity and ease of implementation with redundancy. We first present the current dataspace API syntax and semantics, and then justify and discuss the design considerations.

A. Dataspace API

The core API for the Volpex dataspace communication library consists of calls to add, read and remove data objects to/from an abstract global *dataspace*, with each object identified by a unique *tag* which is an index into the dataspace. The concept of a dataspace is similar to that of a tuplespace in Linda. The main communication calls are as follows:

```
Volpex_put(tag, data)
```

A *Volpex_put* call writes the data object *data* into the abstract dataspace identified with *tag*. Any existing data object with the same tag is overwritten.

`Volpex_read(tag)`
A *Volpex_read* call returns the data object that matches the *tag* in the dataspace.
`Volpex_get(tag)`
A *Volpex_get* call returns the data object that matches the *tag* in the dataspace, and then removes that data object from the dataspace.

Volpex_read and *Volpex_get* calls are identical except that *Volpex_get* also clears the matched data object from the dataspace. Both *Volpex_read* and *Volpex_get* are blocking calls: if there is no matching data object in the dataspace, the calls block until a matching data object is added to the dataspace. A *Volpex_put* call only blocks until the operation is completed. Additional calls are available in the API to retrieve the process Id and the the number of processes, and to initialize and terminate communication with the dataspace server. The basic API is outlined in Table I. In the rest of the paper we use *put*, *get*, and *read* interchangeably with *Volpex_put*, *Volpex_get*, and *Volpex_read*.

int volpex_put (const char tag, int tagSize, const void* data, int dataSize)*
int volpex_get (const char tag, int tagSize, void* data, int dataSize)*
int volpex_read (const char tag, int tagSize, void* data, int dataSize)*
int volpex_getProcId (void)
int volpex_getNumProc(void)
int volpex_init(int argc, char argv[])*
void volpex_finalize(void)

TABLE I
VOLPEX DATASPACE COMMUNICATION API

B. API design considerations

The set of calls in the dataspace API is minimal but is sufficient to simulate basic message passing and shared memory style communication. A data object can be read multiple times until it is removed, and data objects can be overwritten, allowing shared memory style programming. *Read* and *get* operations block when no object with a matching tag exists and the *get* operation clears a data object. These can be used to provide various flavors of synchronization, e.g., barriers, blocking receives, and shared queues for dynamic distribution of work.

The dataspace API is different from the well known Linda system in some significant ways listed as follows:

1) *Single tag:* The parameters for dataspace API calls are a data object and a tag. Data matching is based on a designated tag and associative matching across multiple tuples is not supported. This decision was made for simplicity and efficiency of implementation, without, in our experience, significantly affecting programmability. Our implementation does provide a set of helper functions to generate a unique tag from a set of tuples.
2) *Blocking read calls:* The *read* and *get* calls in the dataspace API are blocking. It is well understood that support for blocking calls is essential to support coordination across processes. Non-blocking calls, where a *get* or *read* returns with no action if no matching object exists, make programming easier in some contexts. An example is a master-worker scenario where a worker checks multiple queues for tasks to execute. Additional non-blocking *read/get* calls can be supported with redundancy, and are being considered as an extension of this work.
3) *Single assignment puts:* Some languages allow a data object to be written only once and not overwritten. This has some desirable properties from software design and implementation perspectives. However, re-assigning to the same tag is essential to easily simulate unstructured shared memory programs. Hence, a multiple assignment model was selected.
4) *Process creation:* There is no support for process creation analogous to Linda *eval* call as process creation and management is done externally.

IV. EXECUTION MODEL

The basic semantics of the communication operations are straightforward as listed in the discussion of the API above. However, managing redundant communication requests is a significant challenge. The key problem is that a logical call (with side effects) may be executed repeatedly or executed at a time when the state of the dataspace is not consistent with canonical execution. For example, what action should be taken if a late running process replica issues a *get* or *read* for which the logically matching data object is not available in the dataspace anymore, either because they were removed by another *get* or overwritten by another *put* ?

The guiding principle for the execution model is that the execution results with redundant communication calls must be consistent with normal execution. We will refer to execution with replicated/redundant communication calls, due to process redundancy or process checkpoint-restarts, simply as redundant execution, for brevity. If the parallel application is deterministic, then normal and redundant executions should give the same results. If the parallel application is non-deterministic but individual process execution is repeatable and deterministic (i.e, non-determinism exists because different communication orders are possible in an execution) then redundant execution will return one possible result of normal execution without replication. The major components of the execution model are the following:

1) *Atomicity rule:* The basic *put/read/get* operations are atomic and executed in some global serial order.
2) *Single put rule:* When multiple replicas of a process issue a *Volpex_put*, the first writer accomplishes a successful operation. Subsequent corresponding *Volpex_put* operations are ignored.
3) *Identical get rule:* The first replica issuing a *Volpex_get* or a *Volpex_read* receives the value stored at the time in the dataspace. Subsequently, replicas of the corresponding *Volpex_get* or *Volpex_read* receive the same value, independent of the time at which they are executed.

The execution model can be explained as follows. The process instances that execute the first instance of a logical communication call create a *leading front* of execution representing normal execution without redundancy. The execution model ensures that *a)* the trailing replica communication calls have no side-effects (single put rule), hence they cannot cause incorrect execution of leading replicas by corrupting the dataspace and *b)* trailing replica communication calls are guaranteed to receive the same data objects for read and get calls as the corresponding first communication calls (identical get rule). All process instances execute identically as the effect of communication calls on the processes is identical irrespective of their execution time and application state at that time. Execution proceeds seamlessly in case of process failures and slowdowns, so long as at least one instance of each process exists. This is illustrated in Figure 1.

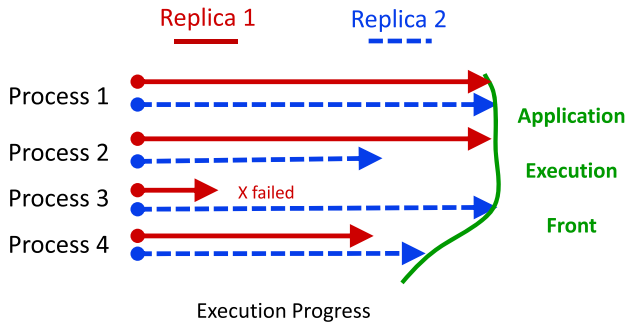


Fig. 1. Application progress is determined by the leading front created by the fastest replica for each process

The fundamental result that we have developed informally is as follows:

Lemma 1: Consider a program with multiple sequential processes communicating exclusively with Volpex dataspace API. Assume that the communication implementation follows the atomicity, single put, *and* identical get *rules. Then any result produced by redundant execution is identical to one of the possible results of normal non-replicated execution.*

As discussed earlier, redundancy may be caused by explicit replicated processes or independent checkpoint-restarts of processes. Non deterministic programs are allowed as long as individual process replicas behave identically and deterministically. Hence the result may not hold, for instance, if a process employs random numbers or has an intermittent software bug. Also, an implicit assumption is that the program does not cause external side effects, e.g., as a result of file or network I/O. No redundancy or checkpoint-restart scheme can work without these conditions. However, in many cases the programs can be rewritten to avoid such problems when they exist, and the dataspace API can assist in the process by providing synchronization mechanisms. A formal proof is omitted for brevity.

V. Implementation

The communication API is implemented with the client server paradigm. The processes connect to a dataspace server that services communication requests. The main challenge is in implementing the execution model with support for redundant processes. We discuss the dataspace server design followed by a discussion of some of the key implementation issues and integration with the BOINC middleware.

A. Dataspace server design

The implementation of the Volpex dataspace API must conform to the execution semantics discussed in Section III. The atomicity rule is satisfied by a single threaded server that processes one client request at a time. In order to satisfy the *single put* and *identical get* rules of the execution model, additional machinery is needed. Each logical communication call (*put, get,* or *read*) is uniquely identified by the pair: *(process_id, request_number)*, where *request_number* is the current count in the sequence of requests from a process. When a communication call is issued by a process, the *process_id* and *request_number* are appended to the message sent to the dataspace server to service the request. For replicated calls corresponding to the same logical call, the *(process_id, request_number)* pairs are identical. This allows the identification of a new call and subsequent replicated calls.

The server implementation maintains the current *request_number* for each process, which is the highest request number served for that process so far. The server also maintains two logically different pools of storage as shown in Figure 2.

- *Dataspace table:* This storage consists of the logically "current" data objects indexed with tags.
- *Read log buffer:* This storage consists of data objects recently delivered from the dataspace server to processes in response to *get* and *read* calls. Each object is uniquely identified by *(process_id, request_number)*.

When a communication API call is executed in a process, a message is sent to the dataspace server consisting of the type and parameters of the call and *(process_id, request_number)* information. A request handler at the server services the call as follows:

- *Put:* If the request number of the call is greater than the current request number for the process (a new put), the data objected indexed with the tag is added to the dataspace table. If the request number of the call is less than or equal to the current request number (a replica put for which the data object must already exist on the server), no action is taken.
- *Get or Read:* If the request number of the call is greater than the current request number for the process (a new get), then i) the data object matching the tag is returned from the dataspace storage, and ii) a copy of the data object is placed in the read log buffer indexed with *(process_id, request_number)*. Additionally if the call is a *get*, the data object is deleted from the dataspace table

(but retained in the read log buffer).

If the request number of the call is less than or equal to the current request number (a replica get for which the data object must exist in the read log), the data object matching *(process_id, request_number)* is returned from the read log buffer.

The design of the dataspace server is illustrated in Figure 2.

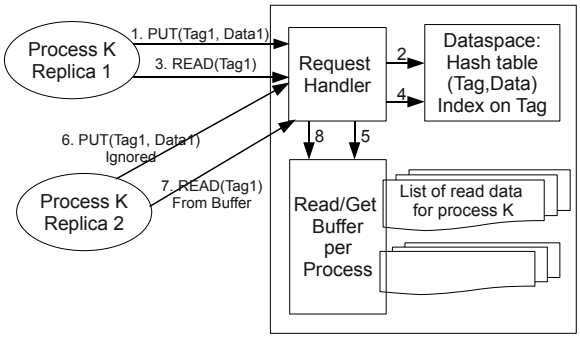


Fig. 2. Volpex dataspace server design

B. Integration with BOINC

The BOINC middleware is widely used for distributed scientific computing with a *bag of tasks* programming model. BOINC runs well on volunteer nodes, because it offers a combination of application-level checkpointing and redundancy to handle failure and computation errors. However, the BOINC platform does not support communicating parallel programs.

This project has leveraged BOINC for management of task distribution and redundancy, while applying the Volpex dataspace API for inter-task communication. When an application is compiled, it can be linked with the BOINC and Volpex libraries. The BOINC redundancy mechanism is employed to create the desired degree of process replication. However, Volpex can also operate independent of BOINC if processes are scheduled explicitly, e.g., with a cluster scheduler.

C. Optimistic logging

The design presented in section V-A is based on pessimistic logging. Each time a data object is delivered as a result of a *get* or a *read* call, the data object is copied to the log. An optimistic approach to logging minimizes copying by taking advantage of the fact that a copy to the log is only necessary if the location with the corresponding tag is overwritten. Hence the following procedure is followed. When an object is delivered from the dataspace in response to a *read*, the corresponding object is only flagged as having been read. The same action is taken on a *get*, except that the object is flagged as logically removed (but not actually removed). When the object is overwritten with a *put*, only then the data object is copied to the log before being overwritten. The replica *read* and *get* calls are directed appropriately to the main dataspace or the log space. Optimistic logging can lead to significant saving in memory and copying overhead but the logic for directing a *read* or *get* request correctly is somewhat more complex. We have developed implementations for both optimistic and pessimistic approaches. However, the results in this paper are based on pessimistic logging and a study of the tradeoffs of the two approaches is beyond the scope of this paper.

D. Log buffer management

An important consideration in the design of a dataspace server is how long should an object be retained in the read log buffer before it is overwritten? In theory a replica or checkpoint restarted process can be arbitrarily out of date with the current state of execution, and hence clearing any old object from the log buffer can cause a communication operation to fail. In practice, a very old copy is unlikely to be needed. The current dataspace server has a circular read log buffer whose size is specified as a parameter during initialization. When the buffer is full, the oldest entry is overwritten. To improve robustness, a simple scheme that utilizes secondary memory space has also been developed. Whenever the log buffer reaches the maximum limit, the messages that are present in the older half of the buffer are removed and written onto the secondary space. Since the dataspace server implicitly tracks the status of all process replicas, there is room for more sophisticated implementations. For instance, a read buffer log entry could be retained until a fixed number of replicas have accessed the object. In the case of usage of checkpointing, log entries can be deleted once a process checkpoint generation ensures that older log entries will not be needed even in the case of a process failure.

E. Distributed and multithreaded implementations

The current dataspace server is a single-threaded server which multiplexes between various requests from the clients. The design allows a distributed implementation by partitioning the abstract global address space whereby each process or thread has exclusive access to a part of the tag address space. The design for a multithreaded implementation, where threads can service arbitrary requests but ensure consistency, has been developed based on similar Linda implementations. As long as concurrent threads are working on independent tags, the only requirement is atomic access to data structures in some cases, such as lists in the log buffers.

F. Implementation framework

Our communication library is built on C/C++ using TCP Sockets. The data provided by the processes is stored in-memory. The tag and data objects are stored in the form of a hash table indexed with tags. The read log buffer is implemented as a combination of hash table and lists. All data transfers are realized as one-way communication initiated by the client processes. The clients establish a connection with the dataspace server using TCP Sockets before performing any operations. This connection is retained until all the operations on the dataspace are completed. If the connection is interrupted, processes try to reestablish the connection with the server in exponentially increasing time intervals.

VI. Usage and results

The Volpex dataspace communication library has been implemented and deployed. It can be used to execute applications with replicated processes for fault tolerance on clusters or on volunteer nodes with the BOINC framework. Experimentation and validation was done on compute clusters as well as ordinary desktops that constitute a "Campus BOINC" installation at University of Houston. In this section we present results with various benchmarks and applications with two different goals. First, document the usability of the dataspace API for a variety of applications with differing communication patterns, and second, understand the performance implications of the Dataspace API design for execution on volunteer environments. The benchmarks and codes that are used for our results are the following:

1) Latency and bandwidth microbenchmarks.
2) *Sieve of Eratosthenes (SoE)*, a well known algorithm for finding prime numbers. The dataspace API was used to broadcast a block of new prime number to all processes in the parallel implementation.
3) *Parallel Sorting by Regular Sampling (PSRS)*, a well known sorting algorithm. The dataspace API was used for all-to-all communication.
4) *Replica Exchange for Molecular Dynamics (REMD)*, a real world application used in protein folding research [27]. Each node runs a piece of molecular simulation at a different temperature using the AMBER program [28]. At certain time steps, temperature data is exchanged between neighboring nodes based on the Metropolis criterion, in case a given parameter is less than or equal to zero. In our implementation of this code, the dataspace API is used for i) storing process-temperature mapping, ii) synchronization of the processes at the end of each step, iii) identification and retrieval of energy values from neighboring processes, and iv) swapping of temperatures between processes when needed.

Additionally a prototype implementation of *MapReduce*, framework for distributed computing was also developed but is not discussed further in this paper. The Dataspace API is used as the intermediary for data exchange between the processors executing the Map and Reduce phases.

The microbenchmarks, Sieve oF Eratosthenes (SoE), and Parallel Sorting by Regular Sampling (PSRS) codes were developed to understand system behavior when the dataspace server is stressed. Replica Exchange Molecular Dynamics (REMD) and MapReduce from Google represent applications with low to moderate degree and volume of communication that are considered suitable for Volpex. In all cases, fault tolerance was achieved by replicating the computation. For the results presented in this paper, the codes were executed on two distinct environments, but with the same dataspace server.

a) Traditional Cluster:: A 300+ node "Atlantis" cluster composed of Itanium2 1.3GHz dual core nodes with 4GB of memory running RedHat Linux. A traditional scheduler (Torque) is used that ensures that only one application runs on a set of nodes and exactly 1 process executes on a node.

b) BOINC Volunteer environment:: A volunteer computing environment managed by BOINC middleware consisting of a cluster, desktops from a UH Computer Science Lab, desktops from a UH Physics Lab, and individual volunteer nodes on campus. The nodes are in active use for other tasks but available to BOINC/Volpex when idle. The cluster has single core AMD MP2000 series nodes with 2GB memory running Linux. The most common configuration of a lab PC was a 1.86GHz Intel x86 processor with 2GB of memory. Computer science lab PCs were running Windows XP and most Physics Lab PCs were running Unix. The default client side BOINC setting that allows *up to 2 processes per node* was used. BOINC picks execution nodes randomly and it was verified that virtually all experiments employed a mix of nodes from the cluster and the two desktop PC pools.

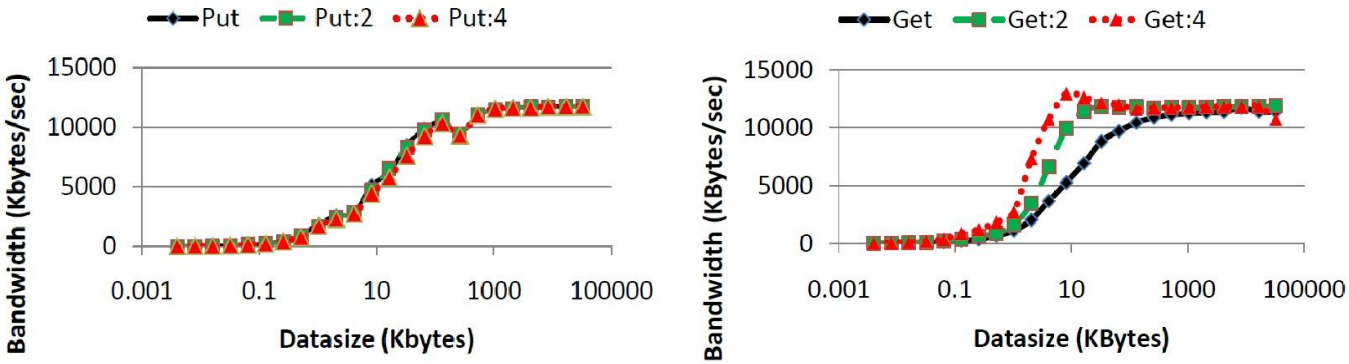
The dataspace server was running on an AMD Athlon 2.4GHz dual core machine with 2GB memory running Ubuntu 9.01. The server and client nodes were on different subnets that are part of a 100Mbps LAN on UH campus. This is the case even for experiments with clients on the traditional cluster.

A. Benchmarking of API calls

In the first set of experiments, we recorded the time taken to execute the different API calls with varying message sizes and varying degree of replication. The effective bandwidth delivered by the server in response to *put* operations from a cluster node is presented in Figure 3(a). Note that the bandwidth presented is the aggregate bandwidth delivered by the server in response to all clients in case of replication.

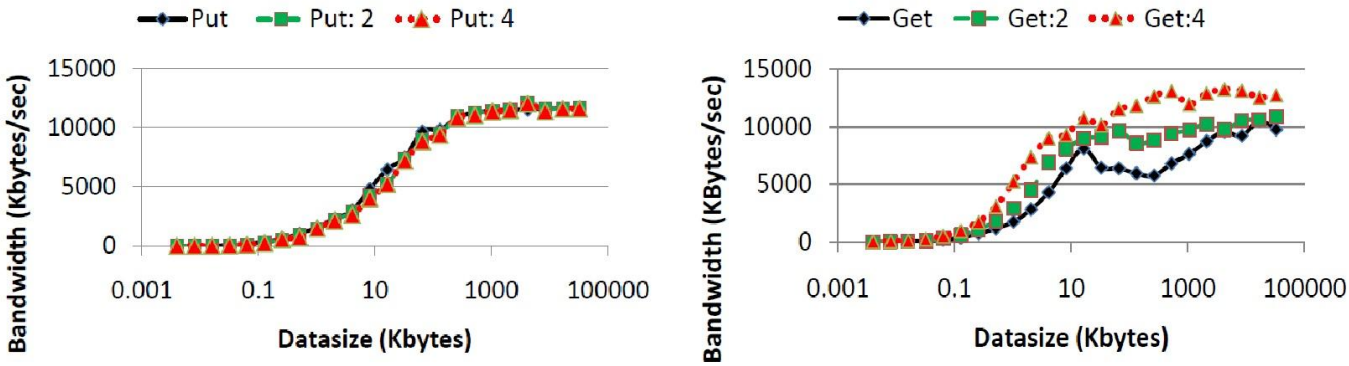
We observe that the general bandwidth trend is typical of a 100Mbps LAN environment. The effective bandwidth increases with the size of the data object but flattens out around 12MBytes/sec (or 96Mbps), which is just below the network capacity of 100Mbps. Hence, the system overhead is not significant. The figure also shows that the effective bandwidth with 2 and 4 replicated processes is virtually identical to that without replication except for a very slight reduction in the midrange of message sizes. It is instructive to recall how replicated *put* operations work. The first *put* actually transfers the data object over the network, and replica *put* calls are returned without any data transfer. Thus, the total network traffic does not increase significantly with replication. Hence, it is not surprising that the effective bandwidth delivered by the server is not significantly affected. The slight reduction is attributed to the overhead of processing of *put* calls from other replicas.

The results for the delivered bandwidth for *get* operations on a cluster are presented in Figure 3(b). We omit the results for *read* operation as they are virtually identical to those for the *get* operation. Without replication, the performance of *get* operations is very similar to the performance of *put* operations and the same discussion applies. However, the behavior with replicas is very different. It is instructive to



(a) PUTs (b) GETs

Fig. 3. Aggregate Bandwidth for PUT and GET operations with and without replicas on a cluster



(a) PUTs (b) GETs

Fig. 4. Aggregate Bandwidth for PUT and GET operations with and without replicas on a Volunteer environment

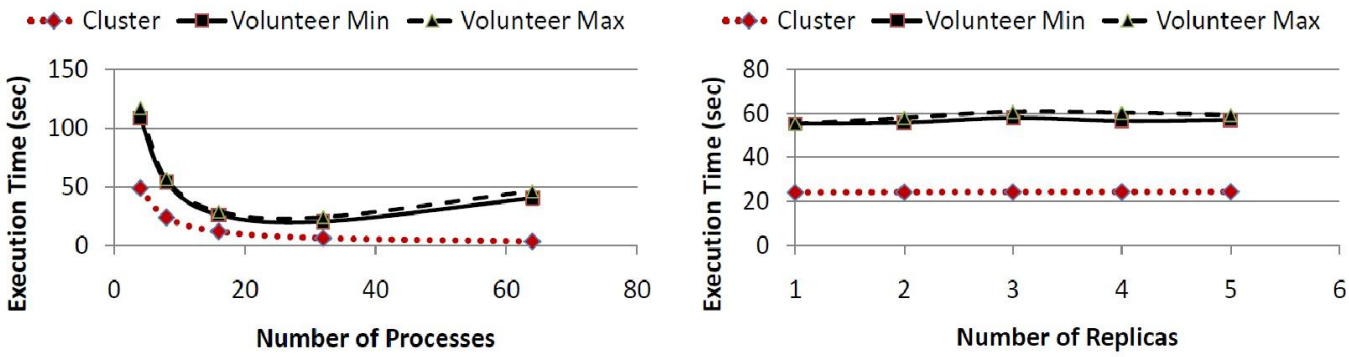
recall that replicated *get* operations are handled very differently from *put* operations. Each replicated *get* call leads to the entire data object being transferred from the server to a client replica. Hence, for a degree of replication of k the network traffic for *get* calls increases by a degree of k while it remains unchanged for *put* operations. Figure 3(b) shows that the aggregate bandwidth delivered by the server *increases* significantly with replication except for very high message sizes where the bandwidth is limited by the network capacity. The server is able to register a higher bandwidth as 2 and 4 replicas imply that the aggregate rate at which the data is being demanded by the clients increases by a factor of 2 and 4, respectively.

Figure 4 presents results from the evaluation of performance of *put* and *get* operations on volunteer environment nodes. The results are virtually identical for *put* operations and slightly lower on average and less predictable for *get* operations. The fact that the results are not significantly different for cluster and volunteer nodes is not surprising since the dataspace server and the 100MBps LAN through which the clients and the dataspace server are connected are identical. The key difference that the cluster hosts are dedicated and faster is not a major factor in this scenario.

B. Sieve of Eratosthenes

We study the SoE program to gain more understanding of performance aspects of employing dataspace for computing. In SoE, prime numbers are identified by eliminating the multiples of discovered prime numbers. In the parallel implementation employed, a block of newly identified prime numbers is broadcast to all processors for elimination of all its multiples. For broadcast with the dataspace API, one process executes a *put* while all other processes issue a corresponding *read*. A low block size of 10 prime numbers was selected for experiments; a lower block size implies frequent *put* and *read* operations of small objects interspersed with small computation blocks making the application latency sensitive. However, a low block size also minimizes the delay in the processing pipeline. The results are shown in Figure 5.

We report a single execution time reading for cluster nodes, but the max and min for 5 runs for the volunteer pool, as the execution time can vary significantly on a volunteer pool. We observe from Figure 5(a) that the execution time for volunteer nodes is a factor of 2 or more higher than cluster nodes. The reasons for this are i) volunteer nodes are somewhat slower



(a) Scalability (b) Replication

Fig. 5. Performance of Sieve of Eratosthenes (SoE)

than cluster nodes, ii) cluster nodes are connected to each other by a low latency high bandwidth network, and iii) up to 2 processes can execute on a single volunteer node while 1 process per node policy is enforced for cluster nodes by the scheduler.

We also observe from Figure 5(a) that SoE scales well on a cluster at least up to the maximum 64 processors tested. However, the performance scales only up to 32 processors on the volunteer pool and then deteriorates. The computation work between consecutive *read* operations is lower for large number of nodes making the execution more latency sensitive. Cumulative impact of heterogeneity in a volunteer pool is also higher for more processors.

Figure 5(b) shows performance of 8 process SoE with replication for fault tolerance. We observe that the performance on the cluster or the volunteer pool is not affected significantly with replication. As the application is latency bound, adding replicas does not impact performance much. This would change if the increased traffic overwhelms the dataspace server, which was observed at significantly higher degrees of replication than what is reported here.

C. Parallel Sorting by Regular Sampling

The Parallel Sorting by Regular Sampling (PSRS) algorithm provides parallel sort operations on distributed data sets. The algorithms consists of three main steps: the selection of pivot points, an all-to-all style communication redistributing the data using the pivot, followed by a local sort operation. The dataspace server is used as the intermediary for all-to-all communication. PSRS is a communication intensive algorithm that was included to investigate the limits of a volunteer pool. PSRS was used to sort 64 million elements. The results in Figure 6(a) show that PSRS scales up to 32 processes on a cluster and 16 processes on the volunteer pool. Interestingly, the performance on the volunteer pool and the cluster is very similar for small numbers of processors. For PSRS, the performance bottleneck is the single dataspace server and the LAN of 100Mbps. Hence, the overall performance is less sensitive to processor performance. The reason for limited scalability is that the performance gain from parallelizing the local sort operation does not match the increased communication costs as the amount of data in each communication step is fixed and the number of communication operations increases. The deterioration in performance is much higher for the volunteer environment for reasons discussed for the SoE code earlier including the impact of heterogeneity for a larger system.

Process replication is our approach to overcoming the volatile nature of nodes. The performance with replication for an 8 process PSRS is demonstrated in Figure 6(b). There is a steady increase in execution time with increasing degree of redundancy. The reason is that the execution time for *get* and *read* operations can increase significantly when the network traffic and the workload overwhelm the server or the network. The execution becomes slower and less predictable rapidly for the volunteer environment above a replication degree of 3.

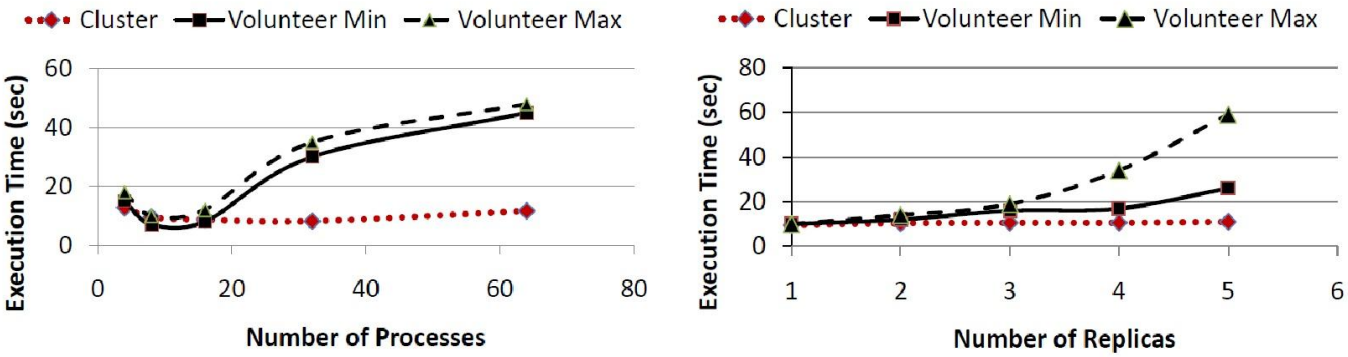
D. Application: Replica Exchange Molecular Dynamics

The Replica Exchange Molecular Dynamics (REMD) formulation [27] tries to overcome the multiple-minima problem by exchanging the temperature of non-interacting temperature replicas of the system running at several temperatures.[1]

The context in which one of the authors of this paper (Cheung) is applying REMD is "crowding" in cell like environments [29, 30]. In REMD, each node runs a piece of molecular simulation at a different temperature using the AMBER program [28]. The number of nodes to simulate the system depends on the system's size and types of interactions. At certain time steps, communication occurs between neighboring nodes. An exchange is initiated based on the Metropolis criterion, in case a given parameter is less than or equal to zero. In our case, the data exchanged is the temperature.

In our implementation of this code, the dataspace API is used for i) storing process-temperature mapping, ii) synchronization of the processes at the end of each step, iii) identification and retrieval of energy values from neighboring processes, and iv) swapping of temperatures between processes when

[1]The term 'temperature replica' has an application specific meaning in the context of REMD, unrelated to Volpex process replicas.



(a) Scalability (b) Replication

Fig. 6. Performance of Parallel Sorting by Regular Sampling (PSRS)

needed. Prior to this collaboration, REMD was implemented on clusters with data exchange performed using a shared file system and file locking.

In the REMD experiments we conducted, each temperature replica represents a process which starts running simulations for a certain temperature. At the end of each step, neighboring temperatures may be exchanged between processes based on the Metropolis criterion. A snapshot of such exchange of temperatures is presented in figure 7, from one of the experiments. It shows how temperatures are exchanged for an execution with 10 temperature replicas and 5 steps. In each step, temperatures that are swapped are highlighted in bold with the same background pattern.

# Step	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	290	300	310	320	**330**	**340**	350	360	370	380
2	290	**300**	**310**	320	340	330	360	350	380	370
3	**290**	**310**	300	340	320	360	330	350	380	370
4	310	**290**	**340**	300	360	320	350	330	370	380
5	310	340	290	360	300	350	320	370	330	380

Fig. 7. REMD Temperature(K) swaps: 10 temperature replicas for 5 steps

Figure 8 presents comparison of application execution time between a volunteer environment with Volpex/BOINC versus execution on a cluster (PERL & File locking) for a small sample REMD run. Execution on volunteer nodes is modestly slower than cluster execution. This application is compute intensive with low communication and the performance difference simply reflects the difference in processing speed of the cluster and volunteer nodes. Process level replication does not impact performance - again an expected result as the low level of communication will not stress the dataspace server. The significant result is that the code can run reliably on ordinary desktops with only a small decline in performance.

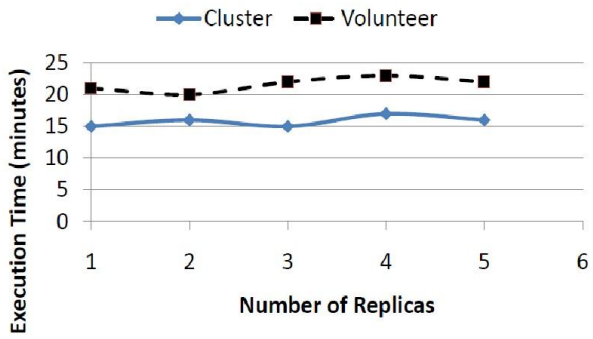


Fig. 8. REMD - Execution on cluster with PERL and on Volunteer nodes

E. Failure behavior

We also evaluated the impact of node failures for all benchmarks and applications when replication was used. In all cases, the application execution time was not negatively impacted with failure of some replicas. In fact failure sometimes led to a slight improvement in performance as it leads to a reduction of communication traffic. Of course, multiple failures will eventually cause the application to fail. Results are omitted for brevity but available in [31].

VII. Conclusions

This paper introduces the Volpex dataspace API that allows efficient Put/Get operations on an abstract global shared memory. An innovative communication model and implementation ensure consistent execution results in the presence of multiple asynchronous invocations of a single communication call due to redundancy for fault tolerance. The system has been integrated with BOINC middleware to provide, to the best of our knowledge, the first comprehensive solution to the reliable execution of communicating parallel codes on volunteer nodes.

The example codes developed make it clear that the framework is applicable to a large class and configurations of parallel applications. The results with primes and sorting programs indicate that the framework can exploit modest

degree of parallelism even on application classes that are considered communication intensive. The experiments and experience with REMD indicates that Volpex can be employed to run large scale practical codes with high computation to communication ratio.

While dedicated compute clusters will always be preferable for many latency sensitive applications, we believe this work fundamentally expands the realm of computing on idle desktops to a much larger class of parallel applications. If a substantial fraction of HPC applications could be executed on shared desktops, the impact will be significant as the clusters can be dedicated to latency sensitive applications that they are designed for.

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. We would also like to acknowledge the contributions of Keith Crabb at UH Research Computing Center and Tsung-I "Mark" Huang at Texas Learning and Computation Center towards providing the infrastructure for this project. Finally, the anonymous reviewers provided invaluable feedback to improve the paper.

REFERENCES

[1] Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the Condor experience. Concurrency - Practice and Experience **17**(2-4) (2005) 323–356

[2] Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Washington, DC, USA, IEEE Computer Society (2004) 4–10

[3] 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)

[4] Carriero, N., Gelernter, D.: The S/Net's Linda kernel. ACM Trans. Comput. Syst. **4**(2) (1986) 110–129

[5] Kondo, D., Taufer, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating desktop grids: an empirical study. Proceedings. 18th International Parallel and Distributed Processing Symposium (April 2004) 26

[6] Ren, X., Eigenmann, R.: iShare - Open internet sharing built on peer-to-peer and web. In: European Grid Conference, Amsterdam, Netherlands (Feb 2005)

[7] Cirne, W., Brasileiro, F., Andrade, N., Costa, L., Andrade, A., Novaes, R., Mowbray, M.: Labs of the world, unite!. Journal of Grid Computing **4**(3) (2006) 225–246

[8] Litzkow, M., Tannenbaum, T., Basney, J., Livny, M.: Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department (April 1997)

[9] : http://www.almaden.ibm.com/cs/tspaces/

[10] Noble, M.S., Zlateva, S.: Scientific computation with javaspaces. In: HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking, London, UK, Springer-Verlag (2001) 657–666

[11] Zhang, L., Parashar, M., Gallicchio, E., Levy, R.M.: Salsa: Scalable asynchronous replica exchange for parallel molecular dynamics applications. In: ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing, Washington, DC, USA, IEEE Computer Society (2006) 127–134

[12] Xu, A., Liskov, B.: A design for a fault-tolerant, distributed implementation of Linda. In: Proc. Nineteenth International Symposium on Fault-Tolerant Computing (FTCS-19), Chicago, IL (June 1989)

[13] Bakken, D.E., Schlichting, R.D.: Supporting fault-tolerant parallel programming in Linda. IEEE Transactions on Parallel and Distributed Systems **6**(3) (1995) 287–302

[14] Jeong, K., Shasha, D.: PLinda 2.0: A transactional/checkpointing approach to fault tolerant Linda. In: Proceedings of the 13th Symposium on Reliable Distributed Systems, Dana Point, CA, USA (1994) 96–105

[15] Docan, C., Parashar, M., Klasky, S.: DataSpaces: An Introduction and Coordination Framework for Coupled Simulation Workflows. In: Proceedings of the High Performance Distributed Computing (HPDC). (2010) 25–36

[16] Duell, J., Hargrove, P., Roman, E.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. In: Berkeley Lab Technical Report (publication LBNL-54941). (2002)

[17] Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In: 8th IEEE International Symposium on High Performance Distributed Computing. (1999)

[18] Rao, S., Alvisi, L., Vin, H.M.: Egida: An extensible toolkit for low-overhead fault-tolerance. In: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, FTCS 99, IEEE Computer Society (03 1999)

[19] Hursey, J., Mattox, T.I., Lumsdaine, A.: Interconnect agnostic checkpoint/restart in Open MPI. In: HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing, New York, NY, USA, ACM (2009) 49–58

[20] Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinie r, P., Magniette, F.: 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, Washington, DC, USA, IEEE Computer Society (2003) 25

[21] Huang, C., Lawlor, O., V., K.L.: Adaptive MPI. In: Languages and Compilers for Parallel Computing. Volume 2958 of Lecture Notes in Computer Science., Springer (2004) 306–322

[22] Batchu, R., Neelamegam, J.P., Cui, Z., Beddhu, M., Skjellum, A., Dandass, Y.: MPI/FT: 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. (2001) 26–33

[23] Genaud, S., Rattanapoka, C.: 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. (2008) 1–8

[24] Ferreira, K., Riesen, R., Oldfield, R., Stearly, J., Laros, J., Redretti, K., Kordenbrock, T., Brightwell, R.: Increasing fault resiliency in a message-passing environment. Technical report, Sandia National Laboratories (2009)

[25] LeBlanc, T., Anand, R., Gabriel, E., Subhlok, J.: VolpexMPI: an MPI Library for Execution of Parallel Applications on Volatile Nodes. In: Proc. The 16th EuroPVM/MPI 2009 Conference, Espoo, Finland (2009) 124–133 Lecture Notes in Computer Science, volume 5759.

[26] Lin, H., Ma, X., Archuleta, J., Feng, W.C., Gardner, M., Zhang, Z.: DataSpaces: An Introduction and Coordination Framework for Coupled Simulation Workflows. In: Proceedings of the High Performance Distributed Computing (HPDC). (2010) 25–36

[27] Sugita, Y., Okamoto, Y.: Replica-exchange molecular dynamics method for protein folding. Chemical Physics Letters **314** (1999) 141–151

[28] 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)

[29] Homouz, D., Perham, M., Samiotakis, A., Cheung, M.S., Wittung-Stafshede, P.: Crowded, cell-like environment induces shape changes in aspherical protein. Proceedings of the National Academy of Sciences **105**(33) (2008) 11754–11759

[30] Stagg, L., Zhang, S.Q., Cheung, M.S., Wittung-Stafshede, P.: Molecular crowding enhances native structure and stability of Alpha/Beta protein flavodoxin. Proceedings of the National Academy of Sciences **104**(48) (2007) 18976–18981

[31] Kanna, N.: Inter-task communication on volatile nodes. Master's thesis, University of Houston (December 2009)