A Communication Framework for Fault-tolerant Parallel Execution

Nagarajan Kanna[1]*, Jaspal Subhlok[1], Edgar Gabriel[1], Eshwar Rohit[1], and David Anderson[2]

[1] Department of Computer Science, University of Houston
[2] UC Berkeley Space Sciences Laboratory

Abstract. 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 checkpoint-restart and/or process redundancy to make continuous forward progress in such an unreliable environment. A communication model based on one-sided Put/Get calls, pioneered by the Linda system, is a good match as processes can execute their communication operations independently and asynchronously. However, Linda and its many variants are not designed for communicating processes that are replicated or independently restarted from checkpoints. 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 validation of a communication layer for robust execution on volatile nodes. The research leads to a practical way to employ idle PCs for latency tolerant parallel computing applications.

1 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

* This material is based upon the work supported by the National Science Foundation's Computer Systems Research program under Award No. CNS-0834750 and CNS-0834807. 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.

[0] Appears in Proceedings of LCPC 2009

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.

Idle desktops represent a potentially immense but *volatile* resource, i.e., they are heterogeneous and their availability to guest scientific applications can change suddenly and frequently based on the desktop owner's actions. Execution of communicating parallel applications on volatile 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 are 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 volatile 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 checkpointed or redundant execution with the final results guaranteed to be identical to (one of the possible) results with normal execution. Management of checkpointing and redundancy is orthogonal to this process; the communication infrastructure does not need to be informed whether, and to what extent, redundancy or checkpoint-restarts are being employed.

This dataspace communication API is a component of the Volpex framework (Parallel Execution on Volatile 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 paper presents the design, execution model, implementation, and preliminary results for the Volpex dataspace API. The communication framework is being integrated with BOINC framework for volunteer computing and target applications include *Replica Exchange Molecular Dynamics* and *MapReduce*.

2 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, in which processes store and retrieve tuples. 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. This paper reports on development of this functionality, which can be employed to support replicated processes or to service communication requests of processes restarted from local checkpoints.

Several implementations of the MPI specification have focused on deploying fault-tolerance mechanisms. The projects either rely on extending the MPI specification to define the state of MPI objects in case of process failures, e.g. the FT-MPI [15] library. Alternatively they use replication techniques, e.g., in MPI/FT library [16] or integrate some form of checkpoint-restart mechanism in order to provide transparent fault-tolerance to MPI applications. MPICH-V [17], a representative of the last category mentioned, is based on uncoordinated checkpointing and pessimistic message logging. The library stores all communications of the system on reliable media through the usage of a *channel memory*. Volpex MPI [18], developed as part of our work, uses a similar approach but employs sender based logging and supports replicated processes.

3 Dataspace programming model

The programming model we have developed consists of independent processes communicating through an abstract *dataspace*. An important consideration was that the execution model allow seamless execution with multiple and varying number of instances of each process. The design of the dataspace was driven by simplicity and ease of implementation with redundancy. We first present the current dataspace API syntax and semantics. Subsequently, we justify the design decisions made and make a case for some changes in the future.

3.1 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 full API is outlined in Table 1.

3.2 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

Table 1. Volpex dataspace communication API

```
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)
```

tag	Identifies each data object in the dataspace
tagSize	Number of bytes of tag
data	Pointer to data object being read/written
dataSize	Number of bytes of data
volpex_put()	Writes data object with the tag value
volpex_read()	Retrieves data object matching tag
volpex_get()	Retrieves & deletes data object matching tag
volpex_getProcId()	Returns process Id of the current process
volpex_getNumProc()	Returns total number of application processes
volpex_init()	Initialize and connect with dataspace server
volpex_finalize()	Releases all resources and terminates

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 Linda in some ways.

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 exist, 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.

4 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 normal 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, then redundant execution will return one possible result of a 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 they are executed.

The execution model can be illustrated 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, so long as at least one instance of each process exists or is re-created from a checkpoint. 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 execution.*

As discussed earlier, redundancy may be caused by explicit replicated processes or independent checkpoint-restarts of processes. An implicit assumption is that the program does not cause external side effects, e.g., as a result of file or network I/O. It is also assumed that there is no non-repeatable program behavior, e.g., due to bugs or use of a random number generator. No redundancy or checkpoint-restart scheme can work without these conditions. However, non deterministic programs are allowed; results with redundancy are one of the possible results of non deterministic execution. A formal proof is omitted for brevity but is straightforward.

5 Implementation

We first present the basic dataspace server design and then discuss major design and implementation issues and choices.

5.1 Dataspace server design

The implementation of the Volpex dataspace API must conform to the execution semantics discussed in Section 3. 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 1.

- *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 1.

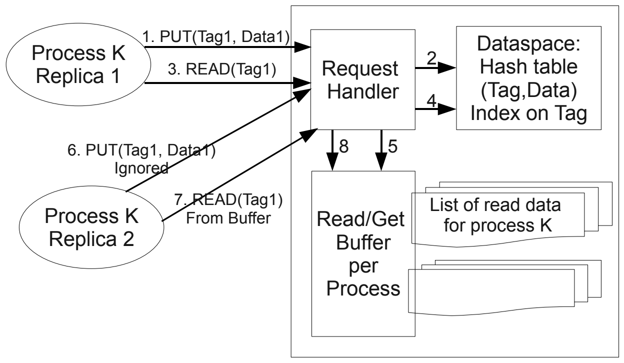

Process K Replica 1
1. PUT(Tag1, Data1)
3. READ(Tag1)
Request Handler
2
4
Dataspace: Hash table (Tag,Data) Index on Tag
6. PUT(Tag1, Data1) Ignored
8
5
7. READ(Tag1) From Buffer
Process K Replica 2
Read/Get Buffer per Process
List of read data for process K

Fig. 1. Volpex dataspace server design

5.2 Optimistic logging

The design presented in section 5.1 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. Future replicas of that call are returned objects from 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. In particular, if a tag is never overwritten, not an uncommon scenario in our experience, no logging is necessary at all. If a data object is read by multiple processes, the memory saving can be proportional to the number of processes. However, the logic for directing a *read* or *get* request correctly is somewhat more complex with optimistic logging. Our current implementation is based on pessimistic logging, and we are developing an optimistic logging based implementation.

5.3 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? 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 a factor in application progress and robustness. 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 deleted. One scheme that is being implemented relies on the use of disk storage for older entries, prior to deletion. 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.

5.4 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.

5.5 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.

5.6 Integration with BOINC

The BOINC middleware is widely used for distributed scientific computing. with a *bag of tasks* programming model. BOINC runs well on volatile 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 on volatile nodes, while applying the Volpex dataspace API for inter-task communication. When an application is compiled, it is linked with the BOINC and Volpex libraries. The BOINC redundancy mechanism is employed to create the desired degree of process replication. However, we currently use BOINC to provide some services while complete integration with BOINC is ongoing.

6 Usage and results

The Volpex dataspace communication library has been implemented and deployed. Experimentation and validation was done on compute clusters as well as ordinary desktops that constitute a “Campus BOINC” installation at University of Houston. Results are presented for clients on a compute cluster for repeatability of experiments.

The dataspace framework has been employed to develop a variety of benchmarks and codes, listed as follows:

1. Latency and bandwidth microbenchmarks.
2. *Sieve of Erastothenes (SoE)*, a well known algorithm for finding prime numbers. The dataspace API was used to broadcast a 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 in the algorithm.
4. *Replica Exchange for Molecular Dynamics (REMD)*, a real world application used in protein folding research [19]. Each node runs a piece of molecular simulation at a different temperature using the AMBER program [20]. 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 to i) store 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.
5. *MapReduce*, a framework for distributed computing from Google. Dataspace is used as the intermediary for data exchange between the processors executing the Map and Reduce phases.

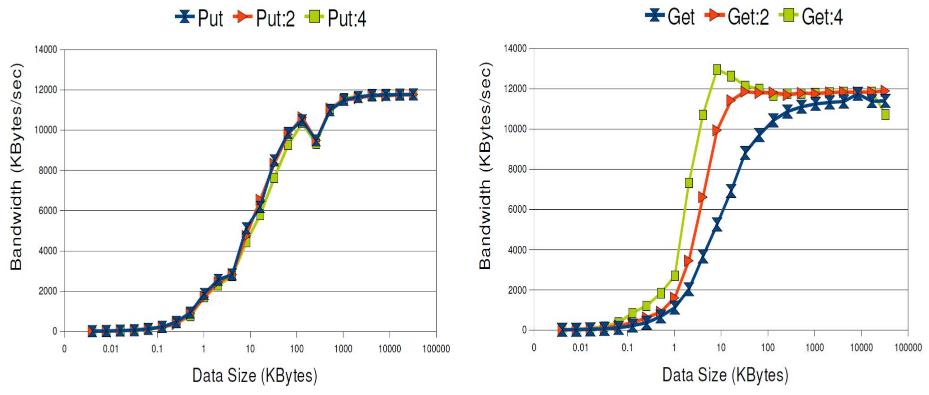
In all cases, fault tolerance was achieved by replicating the computation processes on independent nodes. A full discussion of code development and performance analysis is beyond the scope of this paper, but we discuss sample results from latency/bandwidth benchmarks and SoE code. The codes were executed on the "Atlantis" cluster which has Itanium2 1.3GHz dual core nodes with 4GB of memory running RHEL (5.1). The dataspace server was running on AMD Athlon 2.4GHz dual core with 2GB of memory running Fedora Core 5. The server and client nodes were on different subnets that are part of a 100Mbps LAN on UH campus.

6.1 Benchmarking of API calls

In the first set of experiments, we recorded the time taken to execute the different API calls by the client with varying message sizes and varying degree of replication.

The effective bandwidth delivered by the server in response to *put* operations is presented in Figure 2(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 this 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 the effective bandwidth with 2 and 4 replicated processes. A slight reduction in delivered bandwidth is visible for 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



(a) PUTs (b) GETs

Fig. 2. Aggregate Bandwidth for PUT and GET operations with and without replicas

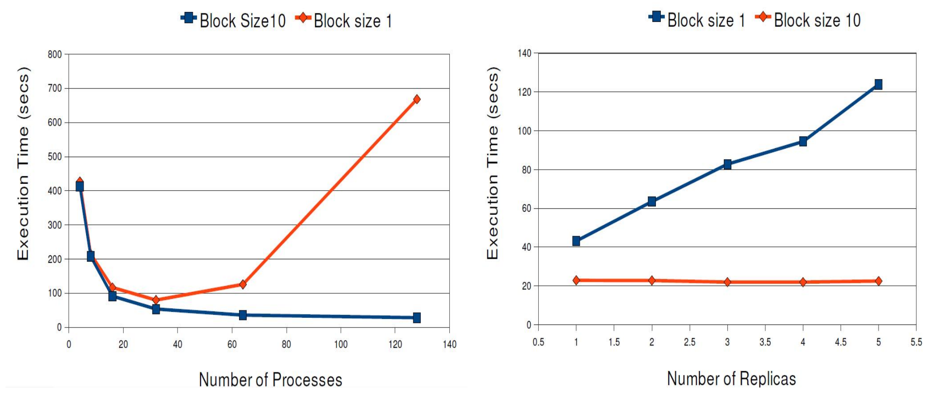
slight reduction is attributed to the overhead of processing of *put* calls from other replicas.

The results for the delivered bandwidth for *get* operations are presented in Figure 2(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 for *get* operations. It is instructive to 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 2(b) shows that the aggregate bandwidth delivered by the server *increases* significantly with replication except for very high message sizes where the bandwidth is (probably) 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.

6.2 Sieve of Erastothenes

We study the 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, a newly identified prime number has to be 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 *read*. A communication optimized version of this program was also developed where a broadcast is done only after a block of prime numbers are discovered. This optimization reduces the number of messages sent by a factor equal to blocksize. The SoE program was executed to discover primes up to 8 billion, without blocking (blocksize = 1) and with a blocksize of 10. The results are shown in Figure 3.



(a) Scalability (b) Replication

Fig. 3. Performance of Sieve of Eratosthenes (SoE)

We observe from Figure 3(a) that the blocked version of SoE scales well up to 128 processors, while the version without blocking scales only up to 32 processors. Frequent broadcast of a single number makes the algorithm without blocking highly latency sensitive, and the dataspace implementation on a LAN does exhibit higher latency than, say, a dedicated cluster. Figure 3(b) shows performance with replication for fault tolerance. Even though a redundancy of 2 is generally sufficient, the execution time was measured with the number of replicas varied from 1 to 5 to gain more insight. Without blocking we see a steady linear increase in execution time by about 20 seconds with each added level of redundancy, which is about half of the execution time without replication. The performance of the blocked version is not affected by replication. The reason is that the unblocked SoE is a communication intensive application where the dataspace server becomes the bottleneck. The execution time increases as increased *read* requests from the replicas overwhelm the dataspace server. While this experiment is designed to stress the dataspace server, it does indicate the need for distributed and multithreaded implementations for increasing the throughput.

6.3 Failure behavior

We also evaluated the impact of node failures on application performance with replicas. 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 [21].

7 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 the employment of checkpoint-restart or redundancy for fault tolerance.

The target of this research is the Volpex execution environment that aims to support efficient execution of communicating parallel programs on volatile idle desktops. The example codes developed demonstrate that the framework can be employed for diverse applications. The overhead of the dataspace framework is low and it delivers the performance and scalability expected on LAN connected nodes, while enabling significant protection against failures.

While dedicated compute clusters will always be preferable for many latency sensitive applications, other loosely coupled parallel applications can gain reasonable performance on ordinary desktops. However, the volatility of desktops is a central problem and the current usage of volatile desktops is limited to embarrassingly parallel (or bag of tasks) or master-slave applications. We believe this work 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. Finally, the dataspace framework is motivated by computing on idle desktops but it can be applied to clusters and other computing environments to increase robustness.

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. Fagg, G.E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J.J.: Process fault-tolerance: Semantics, design and applications f or high performance computing. International Journal of High Performance Computing Applica tions **19** (2005) 465–477
16. 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
17. 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
18. 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 To Appear in Lecture Notes in Computer Science, volume 5759.
19. Sugita, Y., Okamoto, Y.: Replica-exchange molecular dynamics method for protein folding. Chemical Physics Letters **314** (1999) 141–151
20. 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)
21. Kanna, N.: Inter-task communication on volatile nodes. Master's thesis, University of Houston (December 2009)