C-MPI: A DHT Implementation for Grid and HPC Environments

Justin M. Wozniak
Argonne National Laboratory
Argonne, IL, USA

Robert Latham
Argonne National Laboratory
Argonne, IL, USA

Sam Lang
Argonne National Laboratory
Argonne, IL, USA

Seung Woo Son
Argonne National Laboratory
Argonne, IL, USA

Robert Ross
Argonne National Laboratory
Argonne, IL, USA

Abstract

We describe a new implementation of a distributed hash table for use as a distributed data service for grid and high performance computing. The distributed data structure can offer an alternative to existing checkpointing, caching, and communication strategies due to its inherent survivability and scalability. The effective use of such an implementation in a high performance setting faces many challenges, including maintaining good performance, offering wide compatibility with diverse architectures, and handling multiple fault modes. The implementation described here, called Content-MPI (C-MPI), employs a layered software design built on MPI functionality, and offers a scalable data store that is fault tolerant to the extent of the capability of the MPI implementation.

1 Introduction

The objective of this work is to ease the development of survivable systems and applications through the implementation of a reliable key/value data store based on a distributed hash table (DHT) [18]. Borrowing from techniques developed for unreliable wide-area systems, we implemented a distributed data service built with the Message Passing Interface (MPI) [27] that enables user data structures to survive partial system failure. The new service, called Content-MPI (C-MPI), is based on new implementations of the Kademlia [19] distributed hash table, and other hashing techniques.

DHTs offer a reliable distributed storage system for small data objects. They allow for simple queries in a flat key/value namespace. Server nodes in the DHT dynamically join together to create an overlay network that is highly survivable, even in the presence of rapid changes in system membership (churn). DHTs also enable fast lookups even in large systems. Thus, many applications in both Grid and high-performance computing (HPC) benefit from a reusable DHT. Our system is designed to solve problems portably, applying to both environments.

High-performance computing systems continue to grow: currently deployed systems exceed 160,000 cores and systems exceeding 1,000,000 cores are planned. These extremely large systems with correspondingly high failure rates resemble wide-area computing. Currently, applications use checkpointing strategies to save the full state of a computation across fault intervals. However, this method may not be sustainable in the near future as faults rates increase with processor counts [21]. Thus, investigating alternative application models that are inherently survivable is a worthwhile approach.

The checkpoint/restart model typically forces all cooperating tasks to restart in the event of a relatively small fault. This is a behavior that is not scalable in terms of system size and corresponding fault rates, because a fault of unit size requires work to be performed on all participating processors. *Small faults should have small impacts on applications*, resulting in proportionate data movement costs to restore redundancy schemes to a nominal state and normalize the distributed data structures that enable the efficient management of the system.

Our implementation enables the storage of critical application data in a distributed compute-side key/value database, achieving non-volatility through redundancy. Below, we demonstrate that the data structure offers good asymptotic performance at large scale in terms of latency and memory usage, and is capable of enduring multiple faults. The remainder of this paper is organized as follows. In the next section, we consider some recent work applicable to DHTs and MPI. In Sections 3 and 4 we outline design aspects of our system and how it was implemented using

MPI. In Sections 5 and 6, we provide results from experiments with the implementation, and we offer concluding comments in Section 7.

2 Background

DHTs are primarily understood as a hash table that maps keys to distributed buckets: i.e., server nodes. In common parlance, a DHT is understood to have many additional properties, including survivability under rapid churn, performance optimizations for large, wide-area networks, and fast lookup times.

DHTs have been used to develop many practical systems. The Kosha system [2], based on the Pastry DHT [20], establishes a network of NFS services, providing a reliable, distributed filesystem. The target infrastructure for Kosha is the size of a typical university campus network: on the order of tens of thousands of cooperating file servers. The DDNS system [8] implemented a substitute for the Internet Domain Name System (DNS) for distributed name services by storing names and IP addresses as key/value pairs in a Chord [23] network. The DDNS system demonstrated slightly higher remote procedure call counts for DNS lookup operations, but eased DNS administration issues such as administrator error and load balancing. A DHT-inspired routing protocol, Virtual Ring Routing (VRR) [3], routes messages to the addresses of objects or nodes on the network link layer. The technique uses location-independent addresses to reliably locate the destination node without flooding. Most recently, a DHT was proposed in a cluster setting as a mechanism to connect tasks together to form parallel workflows (however, the specifics of the DHT were not described) [29].

Building a fault-tolerant data service requires a framework that provides the programmer a toolkit capable of managing potential fault conditions. In this work, we are interested in the fault handling capabilities offered by the underlying infrastructure, the MPI implementation, and in providing reliability to higher-level application. The fault handling characteristics of MPI have been extensively considered in previous work [14, 16, 17, 26].

In the traditional scientific high-performance computing setting, reliable numerical programming has been addressed by FT-MPI. A primary use case [6] demonstrates how scientific data structures can achieve non-volatility through in-core, parity-based checkpointing, reducing the need for checkpoint operations to disk. In the provided example, an implementation of an iterative numerical algorithm, communication errors were treated with coarse granularity and allowed the programmer to jump back to a known state and proceed with the computation. Users of FT-MPI must perform application-level checkpointing; that is, the application must checkpoint its important state via calls to the FT-MPI system.

Our project differs from these systems in multiple ways. First, we intend to provide a fault-tolerant user data store, and not preserve the state of a computation or in-flight messages. Second, our implementation takes the form of a data service and associated library, not an MPI implementation. Third, since we use an MPI implementation to provide these services, we could potentially use a checkpoint-based fault-tolerant MPI implementation, however, since the DHT provides its own redundancy and other fault tolerant characteristics, we require significantly less from the MPI implementation than is provided by the full-featured solutions above.

3 Design Principles

The distributed hash table was originally targeted to wide-area, Internet-scale systems [18]. DHT systems offer a simple API equivalent to the well-known hash table data structure: $put(\mathbf{key}, \mathbf{value})$ to store an item and $get(\mathbf{key}) \rightarrow \mathbf{value}$ to retrieve a value. DHTs differ from one another in the technique used to *distribute* data over the large number of participating nodes in the system. Peer-to-peer (P2P) DHTs decentralize the organization of the system, creating a reliable key lookup mechanism that is asymptotically fast in terms of the number of participating nodes. This decentralization requires that each peer node be able to participate in the self-organization of the system, enabling robust reassembly in the event of node failure or departure from the system. Additionally, each node is limited to an asympotically small amount of information about the rest of the system.

We chose to investigate MPI as a programming model for the DHT for practical and long-term research reasons. Practically, building the DHT as an MPI library allows it to be linked to codebases from the HPC tradition, developed using C, Fortran, and MPI. These application models would have compatibility issues with typical DHT implementations developed using Java and sockets. Additionally, it offers the high performance available from vendor-specific MPI implementations. Over the long term, we intend to use the DHT to investigate the ability of MPI applications to self-manage their fault tolerance, a feature that is not currently available in existing implementations. This motivation is developed further in Section 4.

3.1 The DHT in the Grid

The DHT and the peer-to-peer model have previously been applied to Grid applications. Building a distributed data structure on the Grid can be accomplished by using a peer-to-peer protocol such as JXTA [15]. This allows diverse applications to dynamically form peer groups and

continue to communicate in the presence of network challenges, such as heterogeneous networks or firewalls. More general messaging technologies such as SOAP [13] may be used as well for unstructured peer-to-peer communication [10].

The DHT may then be used by the application, treating the DHT as a layer [4]. If all cooperating components have access to the DHT, it is a convenient location to store resource information for peer discovery [28]. The DHT itself has been used as a database by GriDB [1], which provides peer-to-peer SQL-like queries, indexed by a DHT. Grid applications can also benefit from filesystem abstractions, enabled by adapters such as FUSE [11]. DFS [5], for example, connects applications to metadata links, backed by storage blocks on existing systems.

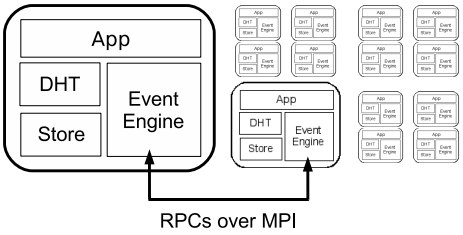

Figure 1. Component model employed in the MPI-based DHT structure.

3.2 The DHT in HPC

Many challenges remain to be overcome for the efficient use of DHTs on real HPC systems. First, existing DHT codes are not designed for HPC, often using programming languages considered unsuitable for HPC, network APIs designed for wide area networks, or system call interception technologies developed primarily as research tools. Second, the DHT concept introduces a layer of indirection in data access that gains scalability at the expense of longer latency; the performance characteristics delivered to real HPC applications remain to be studied. Third, data overwrite is often not supported, because of the challenging consistency issues created by the widely decentralized mechanisms used in many DHTs.

3.3 A Kademlia Implementation in MPI

C-MPI is built on a custom but reusable event-driven system remote procedure call (RPC) library called MPI-RPC that provides a programming model familiar to systems programmers. MPI-RPC allows the programmer to register local functions for invocation by remote processes over MPI. Using this abstraction, user routines may block while waiting for the return from a call or may provide a callback function that will be invoked upon the return. Thus, complex, asynchronous applications may be developed. The components involved in this system are diagrammed in Figure 1. The application and DHT components have access to this system. The DHT additionally has access to a local key/value store that is used to record DHT entries. The whole system is accessible through a simple put/get C-MPI API.

C-MPI provides an abstraction interface over multiple hashing schemes. Multiple implementations are currently provided, including Kademlia and a modulus-based scheme. Thus, a single application written using C-MPI can experiment with the behavior of multiple DHT algorithms without code modification.

The Kademlia DHT algorithm, developed in 2002 by Maymounkov and Mazieres [19], has found practical application in popular systems such as eMule [24]. Kademlia stores user key/value pairs by hashing the keys into a 160-bit linear address space. The space has a concept of closeness measured by the XOR of any two addresses, interpreted as an integer. A Kademlia instance is parameterized by two system-wide parameters: k, the replication level, and α, the parallelism level. Member nodes are assigned locations in this space as well, and each node is responsible for storing a given key/value pair if it is one of the k closest nodes to the hash of the key. Nodes maintain neighbor tables of k addresses for each slice of the address space that differs from the node location by a quantity in $[2^i, 2^{i+1})$. Nodes may identify the k maintainers of a given key by hashing the key, simultaneously contacting the α nodes in the neighbor table closest to that location, and recursively requesting the addresses of nodes closer to the given hash location. For a DHT with N nodes, the tree-like search structure results in a $O(\log(N))$ search time, with local neighbor tables of size $O(\log(N))$.

3.4 MPI Dynamic Processes

C-MPI includes a Kademlia implementation which launches each DHT member node and client exist as separate processes which connect and communicate using MPI. The DHT member nodes maintain *hub* members which advertise the existence of the DHT using the MPI-2 name service functionality, publishing pre-defined names using `MPI_Publish_name()`. After being discovered by other member nodes, the hub nodes allow new member nodes to join the system by accepting new connections and referring these newly connected nodes to other nodes in the

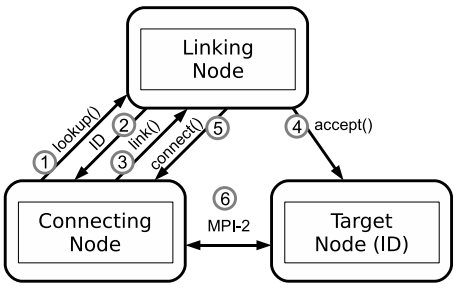

Figure 2. Three-way RPC handshake

hub's neighbor table. The new node is then able to utilize the Kademlia algorithm to integrate with the existing DHT, build up its neighbor table, and update existing neighbor tables by connecting to one node after another.

As shown in Figure 2, the connection process involves six steps. This process is triggered by the normal DHT startup routines and re-entered as necessary during normal operation as member nodes join and exit the system. In a typical case, a new node is discovered as part of a normal DHT lookup operation. In the MPI context, the target node must be notified in advance by an already connected neighbor that a new node would like to connect to it so that the target node may enter `MPI_Comm_accept()`. The connection technique proceeds as follows:

① The connecting node obtains the DHT-specific ID of the target node to which it should connect as part of a typical search.

② The return value of the *lookup()* call is the ID of a node unknown to the connecting node. This node may have the required data item or be closer to it, depending on the DHT algorithm.

③ The connecting node requests a *link()* RPC from the node from which it obtained this ID.

④ The linking node then issues an *accept()* RPC to the target node.

⑤ The linking node then issues *connect()* RPC to the connecting node.

⑥ These two nodes return success to the linking node and then connect by calling into MPI-2 functionality.

* Using normal MPI-RPC functionality, they exchange ID information, updating each neighbor table.

At this point, the connecting node and the accepting node are available for normal operation and may be used as linking nodes for each other by other DHT nodes.

While this technique relies only on MPI for multiprogramming and does not involve more than one node in any communicator, it has multiple unaddressed fault modes. For example, a connecting node could fail, leaving the accepting node stuck in a call to `MPI_Comm_accept()`. Moreover, implementing MPI name service functionality in a reliable way is a challenging problem in itself [9], and may have a variety of fault modes depending on the implementation.

4 Toward Reliable MPI-based Services

Application codes routinely use MPI for data exchange and parallel programming, but rarely if ever does system software use MPI. A previous study of MPI from the perspective of parallel filesystem development [16] covered three primary topics: service discovery and connection, collectives, and fault tolerance. The MPI name service and dynamic process features were proposed as an infrastructure to automate the integration of a network of file servers and connected clients. Predefined collectives as provided by MPI were shown to be potentially useful in many parallel filesystem operations, such as the creation of objects on multiple servers. Fault tolerance, however, was a more difficult problem.

4.1 The DHT Approach

The DHT model provides a reliable scheme to handle faults in a storage network but relies on underlying fault handling functionality. Existing DHT implementations typically obtain this functionality as provided by Java sockets or a similar API. These network abstractions provide several key features to enable the reliable construction of DHTs, such as timeouts on connection methods and recovery from communication errors - features not currently available to the MPI programmer.

Our work proposes a middleware solution for this problem. First, we isolate fault handling under well-defined layers familiar to developers of networked applications. Second, we provide a reliable store for higher-level systems such as filesystems, databases, or application data structures. If the MPI implementation allow us to isolate and dispose of faulty communicators or respond to other fault cases, we could immediately make use of this functionality in our system. Our reliable store could then be used by a variety of application types. In particular, loosely-coupled applications could be built to use the DHT as a reliable, shared tuple space, similar to the functionality offered by PVM persistent messages. Such a system would allow for

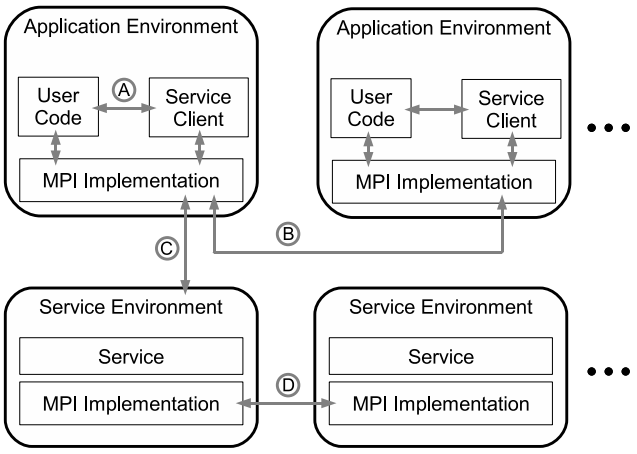

Figure 3. Components and interactions of interest in a generic MPI-based service.

the disposal and restart of compute tasks and DHT member tasks, providing a highly fault-tolerant computing model.

4.2 The DHT as a System Component

Figure 3 illustrates a generic framework for the construction of MPI-based services. In this model, a user code makes use of the service by interacting with a service client, as well as possibly using MPI directly. The system is fragmented into *application environments*, consisting of user computation processes, and *service environments*, consisting of system processes. Service environments may be persistent across multiple executions of user tasks. Application environments correspond to typical compute node settings, whereas service environments correspond to typical file servers or database servers.

Four interactions of interest to MPI are denoted with letters:

A.) The interaction between the user code and the service client. This could be performed via linking, a shell command, or a MPI dynamic process connection.

B.) Interaction among clients. These MPI operations could correspond to collective I/O operations.

C.) Client-server interactions. These connections are likely made via dynamic process operations, but they could be instantiated inside a monolithic MPI execution if services are not persistent.

D.) Server-server interactions. Similar to client-server operations, services may cooperate dynamically or as part of a monolithic communicator.

As a specific example, our DHT implementation makes use of interactions A, C, and D. DHT clients do not interact directly.

The service environments consist of DHT member nodes. During system initialization, DHT member processes are started as unitary MPI programs. These jobs dynamically discover each other using the MPI name service and connect via dynamic process functionality. They then use pairwise intercommunicators [14] to build up a redundant overlay network. User applications make use of the DHT client software, selecting from multiple possible use cases. The service client is currently implemented as a library, and it uses MPI to connect to one or more services. This client-server discovery is based on the MPI name service and results in a connection via dynamic process functionality.

5 Use Case: HPC Setting

In this section, we will study four test cases using a the DHT as a simple database for a hypothetical application, similar to the usage of a Linda system [12]. Consider a user application that performs relatively many small computations and inserts the small results into the DHT-based database to avoid congesting a shared file system with small operations. Each logical key/value pair is to be written by the application exactly once, but since the DHT stores multiple replicas of each key, the output data is relatively persistent compared to the MTTF of a single node. At the end of all computations, the DHT contents could be streamed into a large output file.

5.1 Setup

`TEST-RPC` measures the number of RPC message-pairs used to obtain the MPI ranks responsible for storing a key for the DHT. For each DHT size, one client inserted 4 data items into the DHT at a replica count of 3. The number of RPCs required to perform the DHT lookup to obtain the target site for the insertion was recorded and plotted. These asynchronous search RPCs are parallelizable in accordance with the Kademlia parameter α, as discussed above.

`TEST-TABLE` measures the average size of the neighbor table employed by a member node. For each DHT size, the DHT was started and 4 data items were inserted. The average size of the neighbor table used by each node was recorded.

`TEST-RATE` measures the insertion rate offered by a DHT of increasing size. For each possible DHT size N, a number of clients totaling $N/2$, N, and $2N$ were started.

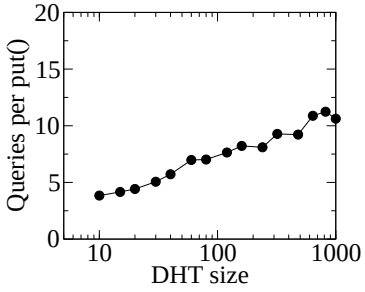

Figure 4. TEST-RPC

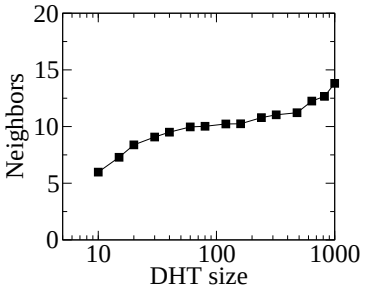

Figure 5. TEST-TABLE

Each then began inserting 1000 20-byte key/value pairs simultaneously (after a call to `MPI_Barrier()`) and the insertion rate was recorded and the average rate was plotted.

TEST-FAULT measures the availability of a given key as the probability that a key may be found in the DHT after a number of member node failures. For each DHT size N, one client inserted N data items at a replica count (k) of 3. Then an increasing number of faults were emulated by the system, implemented by modifying the event engine layer. The number of data items that could be retrieved from the system in this faulty state was recorded and plotted as a percentage.

The 5,832-core SiCortex at Argonne National Laboratory was used in each experiment, which offers small message latency near $1\mu s$ and large message bandwidth of up to 4 GB/s [22]. The SiCortex provides an MPI implementation based on MPICH2. Each of these tests was performed for the Kademlia-based DHT implementation, which was modified to operate within `MPI_COMM_WORLD` due to the lack of MPI-2 dynamic process functionality on the SiCortex. Each test was repeated 5 times and the results were averaged.

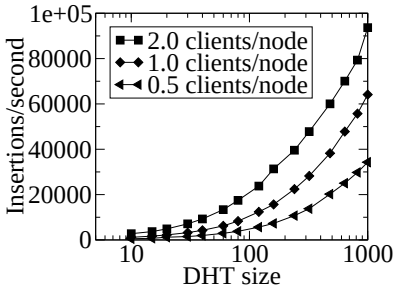

Figure 6. TEST-RATE

5.2 Results

TEST-RPC shows that the data structure is scalable in terms of the number of RPCs required for its operation at sizes up to 1000 DHT member nodes. This indicates that the structure is a promising solution for a simple, distributed, large-scale compute-side database, as it maintains logarithmic performance. TEST-TABLE shows that the neighbor table size is also very small compared to the system size. Each neighbor table entry consists of less than 1 kilobyte of information (not including MPI communicator implementation overhead). Thus, in practice, larger neighbor tables would likely be used to improve performance for

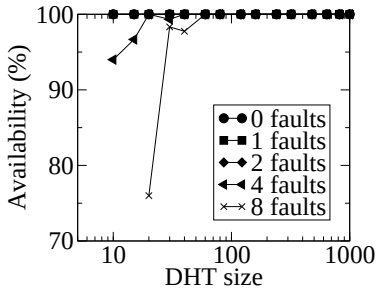

Figure 7. TEST-FAULT

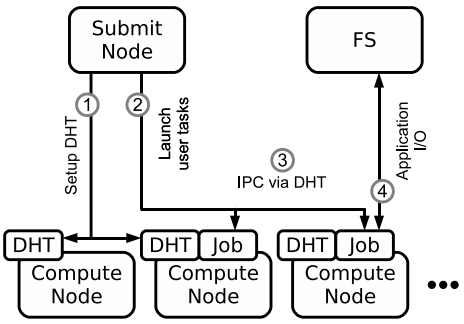

Figure 8. `TEST-CLUSTER` **configuration**

Table 1. Performance results in records/second for cluster setting.

Nodes:	4	8	16	24	30
Records:	200K	200K	352K	432K	480K
FS:	1114	1722	3968	5637	6584
DHT:	2461	4624	9119	12,926	15,533

searches. `TEST-RATE` shows that the insertion rate scales well as expected by the limited number of RPCs reported in `TEST-RPC`. Additional time overhead measured by this test includes the local neighbor table searches and memory operations involved in moving the key/value pairs. This implementation is also resilient to large numbers of emulated faults as shown in `TEST-FAULT`. Although 100% data availability is not guaranteed in cases with more faults than replicas, it is unlikely that all replicas will be hit by a given fault set, and the lookup procedure itself is reliable. Larger systems are capable of very high reliability, and performed perfectly in the batch of tests reported here.

6 Use Case: Cluster Setting

In this section, we will demonstrate the use of the C-MPI DHT in a cluster setting. Many applications cannot be relinked with the C-MPI client library, but the cluster tools are available. As shown in Figure 8, this use case involves multiple steps. First, the user allocates compute nodes from the cluster using ordinary techniques for the given cluster, and submits C-MPI as an MPI job. Second, the batch of single-process user tasks is submitted, and these jobs attach to DHT nodes using local IPC. Third, communication is enabled through the key/value mechanism provided by C-MPI. This is implemented to appear as ordinary filesystem operations in the user script. Fourth, application output is written out to stable storage such as a network file system.

This method is proposed as a way to quickly integrate high-speed IPC into an existing batch application. Typical applications use wrapper scripts to set up access to input and output data sources. In our experimental use case below, only one modification is made to the user script; to change the invocation of the standard `/bin/cp` to the invocation of the C-MPI tool `cmpi-cp`. This tool chunks files into small pieces and distributes them throughout the DHT; for the small files in this case, only one chunk was used per file in addition to a metadata chunk.

6.1 Setup

In this experiment the user script was submitted as described in the cluster setting. The batch of user scripts can communicate in two ways, through the use of small files (FS) or through C-MPI (DHT). The script randomly switches between reading and writing small records of approximately 8 bytes. To demonstrate the additional ability of the C-MPI abstraction to use different underlying DHTs, in this case, a simple hashing scheme was used to locate and store 3 replicas of each record.

This batch was executed on the Breadboard cluster at Argonne National Laboratory. The compute nodes used were a heterogeneous mix including Dual AMD Opteron Dual Core nodes, Dual Intel Xeon Quad Core nodes, and Dual Intel Nehalem Quad Core nodes, all running Linux 2.6.27. In the FS case, the nodes shared access to an NFS server, upon which they exchanged small the records as files. In the DHT case, the processes used C-MPI running on the MPICH2 1.2.1p1 release.

Batches were executed as diagrammed in Table 1. Each node count, from 4 to 30, was associated with a number of records to read and write for communication. For each case, FS or DHT, the script was timed as it performed the given workload; loop overhead and local operations were below 3% of the total run time for any case. The results are shown as record operations/second.

6.2 Results

As shown in the table, both methods show the ability to scale in data access rate with additional processes. This indicates that the NFS server was not overloaded at this scale. However, the DHT exceeded the FS access rate by just over a factor of 2 in each case. Combined with the previous results from the HPC setting, this indicates that the C-MPI system is a promising direction for extremely large clusters.

More generally, these results also indicate that a user-space, local-cluster storage infrastructure built on high-performance technologies may be accessed through standard grid techniques.

7 Conclusion

A reliable key/value data store was proposed as a tool in the development of fault-tolerant systems and applications. A new implementation of the Kademlia DHT on MPI was demonstrated to provide these performance, scalability, and reliability characteristics. Additionally, this system provides a reference point for interesting issues with the development of MPI systems, particularly in fault tolerance. Future work on this system is needed to enable ease of use and compatibility with existing applications. A filesystem interface could be developed by using an adapter such as FUSE [11] or Parrot [25]. This would enable user applications to benefit from the system without modification or user scripts.

The C-MPI implementation is an open source project, available on SourceForge [7].

Acknowledgments

This research is supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contracts DE-AC02-06CH11357. Work is also supported by DOE with agreement number DE-FC02-06ER25777.

References

[1] M. Abdallah and L. Temal. GriDB: A scalable distributed database sharing system for grid environments. In *Proc. Advances in Computer Science and Technology*, 2004.

[2] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proc. SC'04*, 2004.

[3] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: Network routing inspired by DHTs. In *Proc. SIGCOMM*, 2006.

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarcay, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *Proc. SIGCOMM*, 2005.

[5] A. Chazapis, G. Tsoukalas, G. Verigakis, K. Kourtis, A. Sotiropoulos, and N. Koziris. Global-scale peer-to-peer file services with DFS. In *Proc. International Conference on Grid Computing*, 2007.

[6] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *Proc. Symposium on Principles and Practice of Parallel Programming*, 2005.

[7] C-MPI page on SourceForge. https://sourceforge.net/projects/c-mpi.

[8] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a peer-to-peer lookup service. In *Proc. International Workshop on Peer-to-Peer Systems*, 2002.

[9] D. Dewolfs, J. Broeckhove, V. Sunderam, and G. E. Fagg. FT-MPI, fault-tolerant metacomputing and generic name services: A case study. In *Proc. Euro PVM/MPI*, 2006.

[10] G. Fox, D. Gannon, S.-H. Ko, Sangmi-Lee, S. Pallickara, M. Pierce, X. Qiu, X. Rao, A. Uyar, M. Wang, and W. Wu. *Grid Computing: Making the global infrastructure a reality*, chapter Peer-to-peer Grids. Wiley, 2003.

[11] FUSE: Filesystem in userspace. http://fuse.sourceforge.net.

[12] O. Gatibert. YLC, a C++ Linda system on top of PVM. In *Proc. Euro PVM/MPI*, 1997.

[13] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Pearson Education, 1 edition, 2001.

[14] W. Gropp and E. Lusk. Fault tolerance in MPI programs. *J. High Performance Computing Applications*, 18(3), 2004.

[15] JXTA web site. http://jxta.kenai.com.

[16] R. Latham, R. Ross, and R. Thakur. Can MPI be used for persistent parallel services? In *Proc. Euro PVM/MPI*, 2006.

[17] C. Lu and D. A. Reed. Assessing fault sensitivity in MPI applications. In *Proc. SC'04*, 2004.

[18] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7(2), 2005.

[19] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. Workshop on Peer-to-peer Systems*, 2002.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware*, 2001.

[21] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. Distributed Systems and Networks*, 2006.

[22] SiCortex, Inc. SC5832 data sheet. http://www.sicortex.com, 2009.

[23] I. Stoica, R. Morris, D. Karger, M. Kaashock, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *Proc. SIGCOMM*, 2001.

[24] D. Stutzbach and R. Rejaie. Improving lookup performance over a widely-deployed DHT. In *Proc. INFOCOM*, 2006.

[25] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Proc. Workshop on Adaptive Grid Middleware*, September 2003.

[26] R. Thakur and W. Gropp. Open issues in MPI implementation. In *Proc. Asia-Pacific Computer Systems Architecture Conference*, 2007.

[27] The MPI Forum. MPI-2: Extensions to the Message-Passing Interface, 1997.

[28] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi. Peer-to-Peer resource discovery in Grids: Models and systems. *Future Generation Computer Systems*, 23(7), 2007.

[29] E. Walker, W. Xu, and V. Chandar. Composing and executing parallel data-flow graphs with shell pipes. In *Workshop on Workflows in Support of Large-Scale Science at SC'09*, 2009.

Notice