Illinois Institute of Technology
Department of Computer Science

ZHT: a Zero-hop DHT for High-End Computing Environment

A Research Report for PhD Student Qualifying Exam

PhD Student: Tonglin Li
CWID: A20244407
Academic Adviser: Ioan Raicu

1. Introduction

Background

Ken Batcher made a half-serious, half-humorous definition that "A supercomputer is a device for turning compute-bound problems into I/O bound problems", which reveals the essence of modern high performance computing and implies its greatest bottleneck. As the advance of HPC techniques, the peak performances of supercomputers keep soaring, while the I/O bandwidth, especially storage system performance increases much slower. The gap between computing performance and I/O bandwidth is becoming wider.

For next generation supercomputer, namely exscale computers, the challenges are even more radical: almost all of these super computers adopt commodity components and cluster-based design (except for those rare vector machines like the Earth Simulator made by NEC), which means that numerous smaller computer nodes will be used and each of them consumes part of entire I/O.

In the current decades-old architecture of HPC systems, storage is completely segregated from the compute resources and are connected via a network interconnect (e.g. parallel file systems running on network attached storage, such as GPFS [5], PVFS [6], and Lustre [7]). This approach is not able to scale several orders of magnitude in terms of concurrency and throughput, and will thus prevent the move from petascales to exascale. Before new storage architecture and parallel/distributed file system implementations are available, we could not build exascale systems.

As the new Top500 list shows, some of the fastest supercomputers use up to half million processors. The need for building efficient and scalable distributed storage for HEC systems that will scale four orders of magnitude is on the horizon.

Metadata operations on parallel file systems can be inefficient at large scale. Early experiments on the BlueGene/P system at 16K-core scales shows the various costs (wall-clock time measured at remote processor) for file/directory create on GPFS. Ideal performance would be constant. If care is not taken to avoid lock contention, performance degrades rapidly, with operations (e.g. create directory) that took milliseconds on a single core, taking over 1000 seconds at 16K-core scales. [10, 11]

Proposed work

We believe that emerging distributed file systems could co-exist with existing parallel file systems, and could be optimized to support a subset of workloads critical for high-performance computing and many-task computing workloads at exascale. There have been other distributed file systems proposed in the past, such as Google's GFS [13] and Yahoo's HDFS [14]; however these have not been widely adopted in high-end computing due to the workloads, data access patterns, and supported interfaces (POSIX [15]) being quite different. Current file systems lack scalable distributed metadata management, and well defined interfaces to expose data locality

for general computing frameworks that are not constrained by the map-reduce model to allow data-aware job scheduling with batch schedulers (e.g. PBS [16], SGE [17], Condor [18], Falkon [19]).

Future storage systems for high-end computing should support distributed metadata management, leveraging distributed data-structure tailored for this environment. The distributed data-structures share some characteristics with structured distributed hash tables [24], having resilience in face of failures with high availability; however, they should support constant time inserts/lookups/removes delivering low latencies typically found in centralized metadata management. Replication is used to ensure both metadata and data availability.

We are working on delivering exactly such a distributed file system, name FusionFS [30]. FusionFS is a user-level filesystem with a POSIX-like interface through FUSE [27] that runs on the compute resource infrastructure, and enables every compute node to actively participate in the metadata and data management. Distributed metadata management is implemented using ZHT [31], a zero-hop distributed hashtable. ZHT has been tuned for the specific requirements of high-end computing.

Since the limit of report length, this report only focuses on ZHT, my latest work. The previous work I did in last fall is another crucial component of the proposed distributed file system, a FUSE based virtual file system, which will be used as base file system for each node.

We developed ZHT, a zero-hop distributed hash-table, which has been tuned for the specific requirements of high-end computing (e.g. trustworthy/reliable hardware, fast networks, non-existent "churn", low latencies, and scientific computing data-access patterns). The primary goal of ZHT is excellent availability, fault tolerance, high throughput, and low latencies. The report also discusses the use of ZHT as the distributed metadata management in the FusionFS distributed filesystem.

2. Related works

There have been many distributed hash table (DHT) algorithms and implementations proposed over years. In this part I will introduce several existing implementations of DHT.

Chord

The Chord project aims to build scalable, robust distributed systems using peer-to-peer ideas. The basis for much of our work is the Chord distributed hash lookup primitive. Chord is completely decentralized and symmetric, and can find data using only log(N) messages, where N is the number of nodes in the system. Chord's lookup mechanism is provably robust in the face of frequent node failures and re-joins.

On top of Chord's routing layer, a DHash block storage system was built. DHash provides a simple `put` and `get` interface to store and retrieve objects. Recent research has focused on

developing algorithms to efficiently maintain replicas of these objects over long periods to provide availability and durability.

The Chord protocol is one solution for connecting the peers of a P2P network. Chord consistently maps a key onto a node. Both keys and nodes are assigned an *m*-bit identifier. For nodes, this identifier is a hash of the node's IP address. For keys, this identifier is a hash of a keyword, such as a file name. It is not uncommon to use the words "nodes" and "keys" to refer to these identifiers, rather than actual nodes or keys. There are many other algorithms in use by P2P, but this is a simple and common approach.

A logical ring with positions numbered 0 to $2^m - 1$ is formed among nodes. Key *k* is assigned to node successor(*k*), which is the node whose identifier is equal to or follows the identifier of *k*. If there are *N* nodes and *K* keys, then each node is responsible for roughly *K* / *N* keys. When a new node joins or leaves the network, responsibility for *O*(*K* / *N*) keys changes hands.

If each node knows only the location of its successor, a linear search over the network could locate a particular key. This is a naive method for searching the network, since any given message could potentially have to be relayed through most of the network. Chord implements a faster search method. Chord requires each node to keep a "finger table" containing up to *m* entries. The i^{th} entry of node *n* will contain the address of successor $((n + 2^{i-1}) \bmod 2^m)$. With such a finger table, the number of nodes that must be contacted to find a successor in an *N*-node network is *O*(log*N*).

CAN

Like other distributed hash tables, CAN is designed to be scalable, fault tolerant, and self-organizing. The architectural design is a virtual multi-dimensional Cartesian coordinate space, a type of overlay network, on a multi-torus. This d-dimensional coordinate space is a virtual logical address, completely independent of the physical location and physical connectivity of the nodes. Points within the space are identified with coordinates. The entire coordinate space is dynamically partitioned among all the nodes in the system such that every node possesses at least one distinct zone within the overall space.

A CAN node maintains a routing table that holds the IP address and virtual coordinate zone of each of its neighbors. A node routes a message towards a destination point in the coordinate space. The node first determines which neighboring zone is closest to the destination point, and then looks up that zone's node's IP address via the routing table.

Pastry

Although the distributed hash table functionality of Pastry is almost identical to other DHTs, what sets it apart is the routing overlay network built on top of the DHT concept. This allows Pastry to realize the scalability and fault tolerance of other networks, while reducing the overall cost of routing a packet from one node to another by avoiding the need to flood packets.

Because the routing metric is supplied by an external program based on the IP address of the target node, the metric can be easily switched to shortest hop count, lowest latency, highest bandwidth, or even a general combination of metrics.

The hash table's keyspace is taken to be circular, like the keyspace in the Chord system, and node IDs are 128-bit unsigned integers representing position in the circular keyspace. Node IDs are chosen randomly and uniformly so peers who are adjacent in node ID are geographically diverse. The routing overlay network is formed on top of the hash table by each peer discovering and exchanging state information consisting of a list of leaf nodes, a neighborhood list, and a routing table. The leaf node list consists of the $L/2$ closest peers by node ID in each direction around the circle.

In addition to the leaf nodes there is also the neighborhood list. This represents the M closest peers in terms of the routing metric. Although it is not used directly in the routing algorithm, the neighborhood list is used for maintaining locality principals in the routing table.

Tapestry

Tapestry is an extensible infrastructure that provides decentralized object location and routing focusing on efficiency and minimizing message latency. This is achieved since Tapestry constructs locally optimal routing tables from initialization and maintains them in order to reduce routing stretch. Furthermore, Tapestry allows object distribution determination according to the needs of a given application. Similarly Tapestry allows applications to implement multicasting in the overlay network.

Each node is assigned a unique nodeID uniformly distributed in a large identifier space. Tapestry uses SHA-1 to produce a 160-bit identifier space represented by a 40 digit hex key. Application specific endpoints GUIDs are similarly assigned unique identifiers. NodeIDs and GUIDs are roughly evenly distributed in the overlay network with each node storing several different IDs.

Each identifier is mapped to a live node called the root. If a node's nodeID is G then it is the root else use the routing table's nodeIDs and IP addresses to find the nodes neighbors. At each hop a message is progressively routed closer to G by incremental suffix routing. Each neighbor map has multiple levels where each level contains links to nodes matching up to a certain digit position in the ID. The primary i^{th} entry in the j^{th} level is the ID and location of the closest node that begins with prefix (N, j-1)+i. This means that level 1 has links to nodes that have nothing in common, level 2 has the first digit in common, etc. Because of this, routing takes approximately $\log_B N$ hops in a network of size N and IDs of base B (hex: B=16). If an exact ID can not be found, the routing table will route to the closest matching node. For fault tolerance, nodes keep c secondary links such that the routing table has size $c * B * \log_B N$.

Dynamo

Dynamo is a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. Dynamo calls itself as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly. Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Applications that use Dynamo do not require support for hierarchical namespaces or complex relational schema. Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds.

For different problems it faces, Dynamo uses various solutions and achieves very high availability. Some core distributed systems techniques are used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling. For partitioning, it uses consistent hashing, gets Incremental Scalability; for high availability for writes, it uses vector clocks with reconciliation during reads, Version size is decoupled from update rates; for handling temporary failures, it uses Sloppy Quorum and hinted handoff, provides high availability and durability guarantee when some of the replicas are not available; for recovering from permanent failures, it uses anti-entropy with Merkle trees synchronizes, divergent replicas in the background; for membership and failure detection, Dynamo uses Gossip-based membership protocol and failure detection, it preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Kademlia

Kademlia is a DHT-liked peer-to-peer storage system. Kademlia has several features differ from other P2P system.

The configuration information of system doesn't cost an exclusive message to exchange between nodes. They are shown as a part of key queries instead. This makes system-related messages being minimal in Kademlia. Distances between nodes of Kademlia are measured with a new XOR-based metric. Each node has a 160-bit id, the logical distance between any two nodes is decided by a value $d(x, y) = x \oplus y$. (Here $\oplus$ present XOR operation.) XOR is symmetric, allowing Kademlia participants to receive lookup queries from precisely the same distribution of nodes contained in their routing tables.

For keeping nodes' state, Kademlia nodes store contact information about each other to route query messages. For each $\boldsymbol{i}$ between 0 and 160, every node keeps a list of (IP address, UDP port, Node ID) triples for nodes of distance between 2^{i} and 2^{i+1} from itself. The lists are called k-buckets. Each k-bucket is kept sorted by time last seen—least-recently seen node at the head, most-recently seen at the tail. When a Kademlia node receives any message from another node,

it updates the appropriate k-bucket for the sender's node ID. K-buckets effectively implement a least-recently seen eviction policy.

Memcached

Memcached provides a giant hash table distributed across multiple machines. When the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order. Applications using Memcached typically layer requests and additions into core before falling back on a slower backing store, such as a database.

The system uses a client–server architecture. The servers maintain a key–value associative array; the clients populate this array and query it. Keys are up to 250 bytes long and values can be at most 1 megabyte in size. Clients use client side libraries to contact the servers which, by default, expose their service at port 11211. Each client knows all servers; the servers do not communicate with each other. If a client wishes to set or read the value corresponding to a certain key, the client's library first computes a hash of the key to determine the server that will be used. Then it contacts that server. The server will compute a second hash of the key to determine where to store or read the corresponding value.

The servers keep the values in RAM; if a server runs out of RAM, it discards the oldest values. Therefore, clients must treat Memcached as a transitory cache; they cannot assume that data stored in Memcached is still there when they need it. A Memcached-protocol compatible product known as MemcacheDB provides persistent storage.

Cycloid

Cycloid is a constant-degree DHT. It achieves a lookup path length of O(d) with O(1) neighbors, where d is the network dimension and $n = d \cdot 2^d$. It combines Chord and Pastry and emulates a cube-connected-cycles (CCC) graph in the routing of lookup requests between the nodes. Cycloid specifies each node by a pair of cyclic and cubic indices. Like Pastry, it employs consistent hashing to map keys to nodes. A node and a key have identifiers that are uniformly distributed in a $d \cdot 2^d$ identifier space.

C-MPI

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

The following table compares some of the DHT implementations.

	Architecture Topology	Routing Technique	Routing Time(hops)
Chord	Ring	Consistent hashing, finger table	Log(N)
CAN	Virtual multidimensional Cartesian coordinate space, on a multi-torus	Closest destination matching	$O(dn^{1/d})$
Pastry	Hypercube	Closest ID matching	O(logN)
Tapestry	Hypercube	decentralized object location and routing	$O(\log_B N)$
Cycloid	Cube-connected-cycle graph		O(d)
Kademlia	Ring	Dynamic routing table	Log(N)
Memcached	Ring	2-phase hashing client/server	2
C-MPI	Ring	Dynamic routing table	Log(N)
Dynamo	Ring	Consistent hashing	0

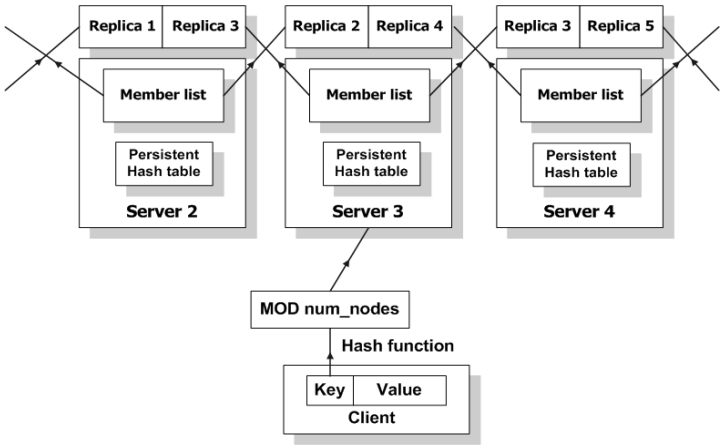
3. ZHT system design and implementation

System Assumptions

It is important to point out that several key features of traditional DHTs are not necessary in HEC, and that if some simplifying assumptions could potentially reduce the number of operations needed per insert/lookup down to a small constant (1 on average). Some of the emphasized key features of HEC are: trustworthy/reliable hardware, fast network interconnects, non-existent node "churn", the requirement for low latencies, and scientific computing data-access patterns. Most HEC environments are batch oriented, which implies that a system that is configured at run time, generally has information about the compute and storage resources that will be available. This means that the amount of resources (e.g. number of nodes) would not increase or decrease dynamically, and the only reason to decrease the allocation is either to handle failed nodes, or to terminate the allocation. By making dynamic membership optional, the complexity of the system can be reduced and a low average number of hops per operation can be achieved. Furthermore, nodes in HEC are generally reliable and have predicable uptime (nodes start on allocation, and nodes shut down on de-allocation). This implies that node "churn" in HEC is virtually non-existent, a property of HEC that should be leveraged to create a more optimized distributed data structure. It is also important to point out that nodes in a HEC system are generally trust-worthy, and that stringent requirements to encrypt communication and/or data would simply be adding overheads. HEC systems are generally locked down from the outside world, behind login nodes and firewalls, and although authentication and authorization is still needed, full communication encryption is wasteful for a large class of scientific computing applications that run on many HEC systems. Most parallel file systems used in HEC communicate between the client nodes and storage servers without any encryption.

Design consideration

The primary goal of ZHT is to get all the benefits of distributed hash tables, namely excellent availability and fault tolerance, but concurrently achieve the benefits of a centralized index where latencies are minimal. The data-structure is kept as simple as possible for ease of analysis and efficient implementation. The membership function ZHT is static, in which every node at bootstrap time has all information about how to contact every other node in ZHT; this assumption is valid because of the batch-scheduled HEC environment. The node id in ZHT will be closely correlated with the network distance between nodes, which can often be inferred from information such as MPI rank or IP address. This will ensure that nodes can easily determine the closest node to communicate with. This network topology aware approach is critical to making ZHT scalable by ensuring that communication is kept localized when performing 1-to-1 communication. The fixed network topology and membership allows the creation of a minimum spanning tree to allow efficient 1-to-all communication.



Hash Functions: There are a many good hashing functions in practice. Each hashing function has a set of properties and designed goals, such as: 1) minimize the number of collisions, 2) distribute signatures uniformly, 3) have an avalanche effect ensuring output varies widely from small input change, and 4) detect permutations on data order. Hash functions such as the Bob Jenkins' hash function, FNV hash functions, the SHA hash family, or the MD hash family all exhibit the above properties. The PI will explore Bob Jenkins' and FNV hash functions, due to their relatively simple implementation, consistency across different data types (especially strings), and the promise of efficient performance.

Membership Table: The proposed hash functions map an arbitrary long string directly to an index value, which can then be used to efficiently retrieve the communication address (e.g. network name, IP address, MPI-rank) from a membership table (e.g. local in-memory array). Depending on the level of information that is stored (e.g. IP - 4 bytes, name - <100 bytes, socket - depends on buffer size), storing the entire membership table should consume only a small (less than 1%) portion of available memory of each node.

Failure Handling: ZHT will gracefully handle failures, by lazily tagging nodes that do not respond to requests repeatedly as failed (using exponential backoff). Once nodes are marked as down,

they are assumed never to return until ZHT is restarted. ZHT will also employ a pro-active node failure detection mechanism by having each $node_i$ pass heartbeat messages with neighboring $node_{i-1}$ and $node_{i+1}$. Upon failure detection, the failed node information would be propagated to all ZHT members through a spanning tree helping all nodes to update the failed node information. The lazy approach is ideal for frequent failures, while the pro-active approach is ideal for rare failures.

Replication: ZHT uses replication to ensure data stored will persist even in face of failures. Newly created data will be pro-actively replicated to nodes in close proximity of the original hashed location. By communicating only with neighbors in close proximity, this approach will ensure that replicas consume the least amount of shared network resources when we succeed to correlate nodes' id with network distance. In order to avoid correlated failures that might cause multiple nodes in close proximity to fail (e.g. a network switch or an I/O node fails), ZHT will also be able to store replicas as far as possible from the source; for example, a replica on $node_i$ could be placed on $node_{(i+n/2)\%n}$. Existing data in ZHT will maintain the minimum replica count by detecting node failures (through periodic neighbor heartbeats), and upon a failure initiate a scan through all the data stored at neighboring nodes to ensure that the minimum replica threshold has not been reached. Furthermore, every node failure would trigger the regeneration of the minimum spanning tree to avoid the failed node (a local operation). Determining the optimal number of replicas is a function of multiple variables, such as number of disks, mean time to failure (MTTF) of each disk, and the mean time to repair (MTTR) of each disk. These variables will determine the value of the mean time to data loss (MTTDL). Several papers have looked at defining ways to compute MTTDL, and they all confirm that at least these variables are critical to computing MTTDL, and some discuss even more complex models that take into account even soft failures (e.g. reading 1 bit with the wrong value).

ZHT is completely distributed, and the failure of a single node does not affect ZHT as a whole. The (key, value) pairs that were stored on the failed node will be replicated on other nodes in response to the failure, and queries asking for data that was on the failed node will be answered by the replicas. In the event that ZHT is shut down (e.g. maintenance of hardware, system reboot, etc), the entire state of ZHT could be loaded from local persistent storage (e.g. the SSDs on each node); note that every change to the in-memory DHT is in fact persisted to a change log to persistent local storage, allowing the entire state of the DHT to be reconstructed if needed. Given the size of memory and SSDs of today, as well as I/O performance improvements in the future, it is expected that this state could be retrieved in seconds to tens of seconds. Once ZHT is bootstrapped, the verification of its nearest neighbors should not be related to the size of the system. In the event that a fresh new ZHT instance is to be bootstrapped, the process is quite efficient in its current static membership form, as there is no global communication required between nodes. For example, in our current prototype which we have tested up to 5832-cores on a SiCortex system, the prototype required only several seconds to completely start and be ready for processing insert or look-up requests. We expect the time to bootstrap ZHT to be insignificant in relation to the cost to batch schedule a large job on a high-end computing system,

which includes node provisioning, OS booting, starting of network services, and perhaps the mounting of some parallel file system.

Persistence: ZHT is a distributed in-memory data-structure. In order to withstand failures and restarts, it utilizes a log-based persistence mechanism, with periodic checkpointing. Any time ZHT is modified, the modification is appended to a file on local persistent storage, and at fixed time intervals the entire state of ZHT is written to the local persistent storage. Upon a restart, the latest checkpoint would be loaded into memory.

Implementation

Application Programming Interface: The application programming interface (API) of ZHT is kept simple and follows similar interfaces for hash tables. The four operations ZHT will support are 1. bool *insert*(key, value); 2. value *retrieve*(key); 3. bool *remove*(key), and 4. bool *broadcast*(key, value). The insert operation will first hash the key to obtain the location of the node where the value will be inserted; existing values will be overwritten; replication occurs asynchronously between the hashed destination and its neighbors. The retrieve operation would return the value from ZHT if it existed. The remove operation would remove the value with the associated key. The broadcast operation would transmit the key/value pair over the edges of the spanning tree with the goal to distribute the key/value pair to all the caches.

Implementation Details: This work is aimed at HEC systems, so it was critical to identify a programming language supported unanimously across all current HEC systems; C/C++ was the best candidate from both a portability and performance perspective. FUSE and GridFTP is implemented in C, while the ZHT is in C++. GridFTP supports a variety of communication protocols, such as UDT and TCP; it supports GSS-API authentication of the control channel and data channel, and supports user-controlled levels of data integrity and/or confidentiality. ZHT used a proprietary binary UDP-based protocol; we are also investigating alternative communication protocols such as TCP, MPI, and BMI.

4. Performance Evaluation

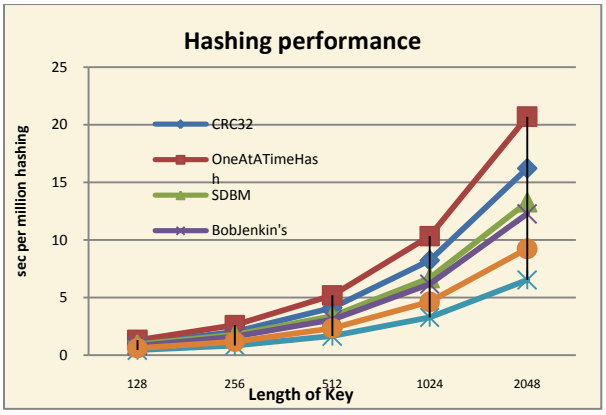
In this section, we evaluate the performance matrices of ZHT, including hashing functions, persistent hash map, different communication methods, throughput, and compare ZHT with other related work to illustrate and prove the scalability and high performance of ZHT.

Hashing Functions

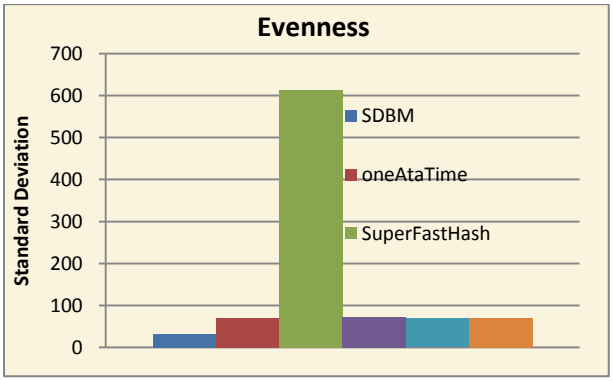
The time spent on hashing keys to nodes is not major of the total cost, but with the time passed, the accumulation could be observed. We investigate some of the usual hash functions for figure out the performance and evenness difference.

As shown below that some hash functions are faster than others, but a more important concern rather than performance is the evenness. Since the worst hash function we investigated has

performance of 0.02ms/hash, and thus negligible compare to other overhead. Meanwhile the evenness is essential to the entire performance.



An ideal hash function should be able to spread keys evenly. Additionally it should not increment the time too rapidly as the length of key increased. We need the hash function to be even to provide a natural load balancing mechanism. But obviously sometimes we can't obtain both performance and evenness. SuperFastHash, for example, as it's named, is very fast, but its evenness is poor. We adopt SDBM hash function mapping keys to destination nodes. As observed in the charts below, SDBM offers a reasonably good performance while gives very good evenness. We measure 1 million keys distribution over 1000 nodes, then find that the standard deviation is only 32.7. This is almost half of most of other hash functions' standard deviation.

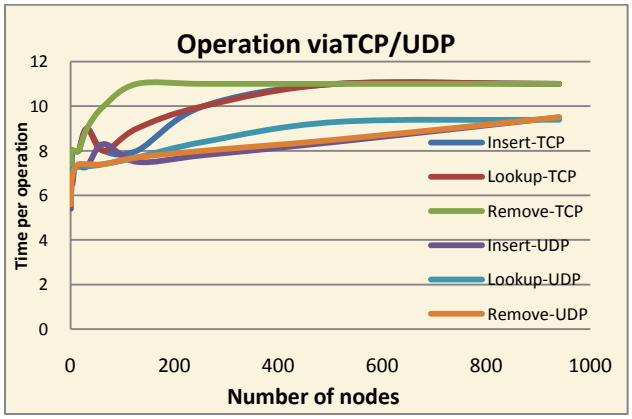


Persistent Hash Map

We adopt Kyotocabinet as local and persistent key-value store. Kyoto Cabinet is a library of routines for managing a database. The database is a simple data file containing records, each is a pair of a key and a value. Every key and value is serial bytes with variable length. Both binary data and character string can be used as a key and a value. In a file based hash table (HashDB) mode on one node, it takes 2955.94 ms to insert, 1940.06 ms to lookup and 2511.77 ms to remove 1 million key-value pairs. Thus less than 3micro sec is taken by per operation. This overhead is negligible compare to the communication time.

Networking

Generally it is considered that TCP offers reliable communication while UDP provides better performance. We implemented ZHT with both TCP (with server returned result state) and UDP (acknowledge message based, which means every time a message is sent, the sender is waiting for an acknowledge message) protocols. As shown that UDP does give a better performance, although the difference between them is not significant. TCP implementation keeps the latency within 11 ms while UDP does it with 9ms. Additionally we note that when the network becomes to saturate, TCP shows a better scalability and reliability. The performance differences among three basic operations (insert, lookup and remove) are very small and tend to be negligible. The time per operation in milliseconds starts at about 6ms at the smallest scales of 2 nodes (1 node scales is artificially fast because it has no network communication), and increases to 11ms to 12ms at the largest scale of 972 nodes. Since ZHT uses a direct 0-hop algorithm and that the majority of the overhead comes from network communication, it is not expected that the time per operation to increase significantly with larger scales.



Throughput and Latency

In our implementation we completed an early prototype of ZHT, which supports three of the four proposed operations, namely insert/retrieve/remove. Prior to engaging in implementing ZHT, we explored the feasibility of using existing DHT implementations (Tapestry [36], Chord [34], Maidsafe-dht [42], and C-MPI [43]), but came to the conclusion that they are all too heavy weight for HEC, were not feature complete, or had significant dependencies to resolve. The preliminary ZHT implementation was implemented in C++, and uses a proprietary binary TCP-based and UDP-based communication protocol. We performed experiments (on the SiCortex SC5832 at ANL, 972 nodes with 5832 cores) to measure the overheads of insert/remove/retrieve. Overheads were significantly lower than the DHTs that were investigated, with about 10ms per operation at modest scales of 5832-cores (as opposed to 30 ms for Chord and 1000 ms for Maidsafe-DHT at 16-core scales). Figure 1 shows the throughput for ZHT for a range of scales from 1 node (6 cores) to 972 nodes (5832 cores). Each client (1 client per node) performs 100K random 132-byte key inserts, 100K retrieves, and 100K removes. The throughputs in operations per second increases near-linearly with scale, reaching 85K ops/sec at 972 nodes; the ideal throughput would have been 156K ops/sec. Also, as shown in the figure, the UDP protocol has

better performance than TCP, which is near 100k ops/sec, and would might scale better in even large scale, nevertheless, based on the volatile propriety of UDP and bottleneck of network, UDP protocol in our implementation will lose performance when the network is saturated, after all, it does not hold a reliable way.

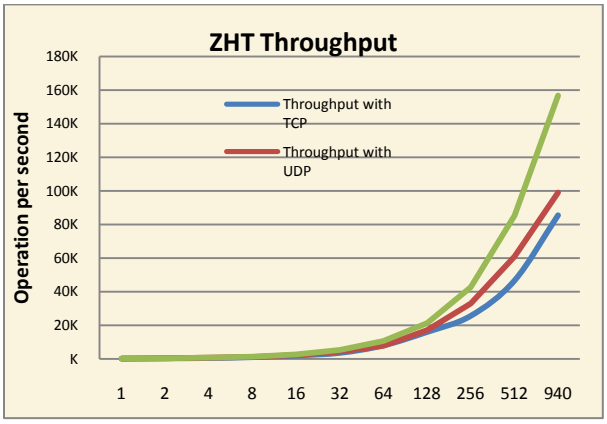
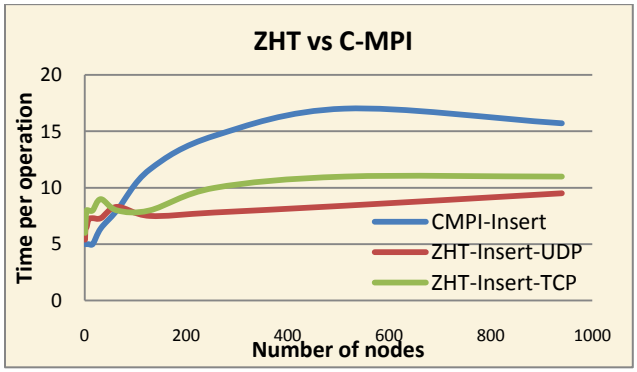


Figure 1: ZHT performance on SiCortex SC5832 on up to 972 nodes (5832-cores)

TCP vs. UDP vs. MPI

We also conduct Content-MPI (C-MPI)[43] experiment. C-MPI is a DHT implementation based on Kademlia DHT algorithm by Argonne National Laboratory with MPI communication mechanism: clients send and route requests to servers, the servers operate (insert, find or remove) on hash table, except each server maintains a neighbor table which does not contain all the server members. The experiment is deployed across 1 to 950 nodes on Sicortex SC5832 with one client and on server process on one node. At the largest scale test, C-MPI has the average insert operation time at 15.8 ms compared with 11 ms and 9.5 ms in ZHT-TCP and ZHT-UDP respectively. At this time, the performance of ZHT overweighs C-MPI's, the conjectured explanation is that the communication latency in MPI costs more than simple TCP or UDP. Additional explanation is that MPI code doesn't scale well when the number of nodes is going on, as shown in the graph, the operation time cost goes nearly linearly from 1 to 200 nodes, however, ZHT with TCP or UDP has a gentle growth, which also reveal that ZHT has good scalability, even in larger scale compared to MPI implementations.



5. Future work

We have many ideas on improving ZHT and on possible use cases where ZHT could make a significant difference in performance or scalability. This section discusses such improvements (e.g. spanning trees), increased scalability studies at hundreds of thousands of core scales (e.g. IBM BlueGene/P, IBM BlueWaters, Cray XT5), and some concrete use cases that we are considering (e.g. FusionFS [cite], MosaStore [cite], Swift [cite], dFalkon [cite]).

ZHT Enhancements

Enhanced proactive failure detection: Existing data in ZHT will maintain the minimum replica count by detecting node failures (through periodic neighbor heartbeats), and upon a failure initiate a scan through all the data stored at neighboring nodes to ensure that the minimum replica threshold has not been reached.

Spanning tree for 1-many communication: An important data dissemination pattern is the 1-many communication pattern (e.g. broadcast). Spanning trees are a great abstraction to implement this 1-many pattern in order to improve performance and reduce hot-spots. A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle. [**Error! Reference source not found.**] Minimum spanning trees are spanning trees that have minimum total weight. Many graph minimum spanning tree algorithms have been published, with the best taking linear time in either a randomized expected case model [**Error! Reference source not found.**] or with the assumption of integer edge weights [**Error! Reference source not found.**]. Since ZHT would have integer edge weights (the approximate distance between nodes in the linear node id space), the spanning tree form an approximate minimum spanning tree closely matching the network topology. Also, due to the membership being static and well defined, all nodes can compute their local independent minimum spanning trees, and all nodes will inherently obtain the same minimum spanning tree. This makes re-computing the spanning tree efficient.

FusionFS

We proposes to the Fusion Distributed File System (FusionFS) optimized for a subset of HPC and MTC workloads. Every compute node will have all three roles, client, metadata server, and storage server. The metadata servers will use ZHT, which will allows the metadata information to be disperse throughout the system, and allow metadata lookups to occur in constant time. FusionFS will be accessible through library calls, and through a standard POSIX interface enabled by FUSE. As many current large-scale supercomputers do not have any persistent local storage on the compute nodes, the initial work will run on in-memory ram-disks that will be used primarily for fast scratch space; once persistent local storage on each node exists, FusionFS could be a persistent distributed file system. The userspace FusionFS implementation interacts with ZHT for metadata information, and then uses GridFTP to transfer files/chunks from remote resources. Note that every compute node has all three functions, client (FUSE), metadata server (ZHT), and data server (GridFTP).

Distributed Metadata Management: FusionFS will use ZHT to implement the distributed metadata management. Every file will have metadata regarding the file (e.g. size, creation date, modification date, permissions, etc) as well as specific metadata about the data locality (e.g. chunk location on remote resources). Directories will be special files containing only metadata about the files in the directory.

Data Indexing: When dealing with massive data collections, one challenge is indexing the material to support re-use and analysis. All of the storage levels will need to support descriptive, provenance [**Error! Reference source not found.**], and system metadata on each file. The proposed solution encompasses metadata management at the same level as data management. We will explore the possibility of using ZHT to index data (not just metadata) based on its content. For this indexing to be successful, some domain specific knowledge regarding the data to be indexed will be necessary. A general solution to data indexing is hard, and certainly not the focus of this proposal. However, data indexing is an important issue as datasets continues to grow and efficient mechanisms to deal with large data collections is growing in importance.

6. Conclusion

The ideas in this report are transformative due to their departure from traditional HEC architectures and approaches, while proposing radical storage architecture changes based on distributed file systems to make exascale computing a reality. ZHT optimized for high-end computing systems is architected and implemented as a foundation in the development of fault-tolerant, high-performance, and scalable storage systems. We performed an extensive performance evaluation of ZHT. We measured the performance of ZHT over TCP and UDP at scales up to 5400-cores on a SiCortex SC5832 supercomputer.

The controversial viewpoints of this work can make exascale computing more tractable, touching every branch of computing in HEC. They will extend the knowledgebase beyond exascale systems into commodity systems as the fastest supercomputers generally become the mainstream computing system in less than a decade; the solutions proposed here will be relevant to the data centers and cloud infrastructures of tomorrow. These advancements will impact scientific discovery and global economic development. They will also strengthen a wide range of research activities enabling efficient access, processing, storage, and sharing of valuable scientific data from many disciplines (e.g. medicine, astronomy, bioinformatics, chemistry, aeronautics, analytics, economics, and new emerging computational areas in humanities, arts, and education). If these ideas materialize, they will transform the storage systems for future HEC systems, and open the door to a much broader class of applications that would have normally not been tractable. Furthermore, the concepts, data-structures, algorithms, and implementations that underpin these ideas in resource management at the largest scales can be applied to new emerging paradigms, such as Cloud Computing.

References

[1] Top500 Supercomputer Sites, Performance Development, http://www.top500.org/lists/2010/11/performance_development, November 2010

[2] T. Hey, S. Tansley, and K. Tolle. The Fourth Paradigm: Data-Intensive Scientific Discovery, Microsoft Research 2009

[3] V. Sarkar, et al. "ExaScale Software Study: Software Challenges in Extreme Scale Systems", ExaScale Computing Study, DARPA IPTO, 2009

[4] B. Obama. "A Strategy for American Innovation: Driving Towards Sustainable Growth and Quality Jobs", National Economic Council, 2009

[5] F. Schmuck, R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," FAST 2002

[6] P. H. Carns, W. B. Ligon III, R. B. Ross, R. Thakur. "PVFS: A parallel file system for linux clusters", Proceedings of the 4th Annual Linux Showcase and Conference, 2000

[7] P. Schwan. "Lustre: Building a file system for 1000-node clusters," Proc. of the 2003 Linux Symposium, 2003

[8] I. Wladawsky-Berger. "Opinion - Challenges to exascale computing", International Science Grid this Week, April 2010; http://www.isgtw.org/feature/opinion-challenges-exascale-computing

[9] Wikipedia contributors. "Achilles' heel." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 8 Jun. 2010. Web. 29 Jun. 2010

[10] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Toward Loosely Coupled Programming on Petascale Systems," IEEE SC 2008

[11] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, M. Wilde. "Design and Evaluation of a Collective I/O Model for Loosely-coupled Petascale Programming", IEEE MTAGS08, 2008

[12] E.N. Mootaz Elnozahy, et al. "System Resilience at Extreme Scale", Defense Advanced Research Project Agency (DARPA), 2007

[13] S. Ghemawat, H. Gobioff, S.T. Leung. "The Google file system," 19th ACM SOSP, 2003

[14] A. Bialecki, M. Cafarella, D. Cutting, O. O'Malley. "Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware", 2005

[15] J.S. Quarterman, S. Wilhelm, "UNIX, POSIX, and Open Systems: The Open Standards Puzzle", Addison-Wesley, Reading, MA, 1993

[16] B. Bode, et al. "The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters", Usenix, 4th Annual Linux Showcase & Conference, 2000

[17] W. Gentzsch. "Sun Grid Engine: Towards Creating a Compute Power Grid," IEEE CCGrid, 2001

[18] D. Thain, T. Tannenbaum, M. Livny, "Distributed Computing in Practice: The Condor Experience", Concurrency and Computation: Practice and Experience, 2005

[19] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde. "Falkon: A Fast and Light-weight tasK executiON Framework," IEEE/ACM SC 2007

[20] A. Birrell, M. Isard, C. Thacker, T. Wobber. "A design for high-performance flash disks", Operating Systems Review, 41(2):88–93, 2007

[21] Intel News Release. "Intel, STMicroelectronics deliver industry's first phase change memory prototypes", http://www.intel.com/pressroom/archive/releases/20080206corp.htm, 2008

[22] W. Jiang, C. Hu, Y. Zhou, A. Kanevsky. "Are disks the dominant contributor for storage failures? A comprehensive study of storage subsystem failure characteristics", In USENIX Conference on File and Storage Technologies (FAST), pages 111–125, 2008

[23] Intel Product Manual. "Intel® X25-E SATA Solid State Drive", http://download.intel.com/design/flash/nand/extreme/319984.pdf, 2009

[24] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. "Looking up data in P2P systems", Communications of the ACM, 46(2):43–48, 2003

[25] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, B. Tierney, "File and object replication in data grids," ACM HPDC-10, 2001

[26] S. Podlipnig, L. Böszörmenyi. "A survey of Web cache replacement strategies", ACM Computing Surveys (CSUR), Volume 35 , Issue 4, Pages: 374 – 398, 2003

[27] "Filesystem in Userspace", http://fuse.sourceforge.net/, 2011

[28] W. Vogels, “Eventually consistent,” ACM Queue, 2008.

[29] I. Raicu, I. Foster, Y. Zhao. “Many-Task Computing for Grids and Supercomputers”, IEEE MTAGS08, 2008

[30] FusionFS: Fusion distributed File System, http://datasys.cs.iit.edu/projects/FusionFS/index.html, 2011

[31] ZHT: Zero-Hop Distributed Hash Table for High-End Computing, http://datasys.cs.iit.edu/projects/ZHT/index.html, 2011

[32] P. Maymounkov, D. Mazieres. “Kademlia: A Peer-to-peer Information System Based on the XOR Metric”, In Proceedings of IPTPS, 2002

[33] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, “A scalable content-addressable network,” in Proceedings of SIGCOMM, pp. 161–172, 2001

[34] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", ACM SIGCOMM, pp. 149-160, 2001

[35] A. Rowstron and P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems,” in Proceedings of Middleware, pp. 329–350, 2001

[36] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment", IEEE Journal on Selected Areas in Communication, VOL. 22, NO. 1, 2004

[37] B. Fitzpatrick. “Distributed caching with memcached.” Linux Journal, 2004(124):5, 2004

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. “Dynamo: Amazon’s Highly Available Key-Value Store.” SIGOPS Operating Systems Review, 2007

[39] H. Shen, C. Xu, and G. Chen. Cycloid: A Scalable Constant-Degree P2P Overlay Network. Performance Evaluation, 63(3):195-216, 2006

[40] Ketama, http://www.audioscrobbler.net/development/ketama/, 2011

[41] Riak, https://wiki.basho.com/display/RIAK/Riak, 2011

[42] Maidsafe-DHT, http://code.google.com/p/maidsafe-dht/, 2011

[43] C-MPI, http://c-mpi.sourceforge.net/, 2011

[44] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, K. Iskra, P. Beckman, I. Foster. “Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers”, SciDAC09, 2009

[45] C. Docan, M. Parashar, S. Klasky. "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows", ACM HPDC 2010

[46] H. Stern. "Managing NFS and NIS". O’Reilly & Associates, Inc., 1991

[47] P.J. Braam. "The Coda distributed file system", Linux Journal, #50, 1998

[48] D. Nagle, D. Serenyi, A. Matthews. "The panasas activescale storage cluster: Delivering scalable high bandwidth storage". In SC ’04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004

[49] Microsoft Inc. "Distributed File System", http://www.microsoft.com/windowsserversystem/dfs/default.mspx, 2011

[50] GlusterFS, http://www.gluster.com/, 2011

[51] Isilon Systems. "OneFS", http://www.isilon.com/, 2011

[52] "POHMELFS: Parallel Optimized Host Message Exchange Layered File System", http://www.ioremap.net/projects/pohmelfs/, 2011

[53] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, E. Cesario. "XtreemFS - a case for object-based storage in Grid data management". VLDB Workshop on Data Management in Grids, 2007

[54] Y. Gu, R. Grossman, A. Szalay, A. Thakar. “Distributing the Sloan Digital Sky Survey Using UDT and Sector,” e-Science 2006

[55] CloudStore, http://kosmosfs.sourceforge.net/, 2011

[56] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, C. Maltzahn. "Ceph: A scalable, highperformance distributed file system". In Proceedings of the 7th OSDI, 2006

[57] W. Xiaohui, W.W. Li, O. Tatebe, X. Gaochao, H. Liang, J. Jiubin. “Implementing Data Aware Scheduling in Gfarm Using LSF Scheduler Plugin Mechanism,” GCA05, 2005

[58] X. Wei, Li Wilfred W., T. Osamu, G. Xu, L. Hu, J. Ju. “Integrating Local Job Scheduler – LSF with Gfarm,” ISPA05, vol. 3758/2005, 2005

[59] MooseFS, http://www.moosefs.org/, 2011

[60] D. Thain, C. Moretti, J. Hemmes. “Chirp: A Practical Global Filesystem for Cluster and Grid Computing,” JGC, Springer, 2008

[61] S. Al-Kiswany, A. Gharaibeh, M. Ripeanu. "The Case for a Versatile Storage System", Workshop on Hot Topics in Storage and File Systems (HotStorage’09), 2009

[62] P. Druschel, A. Rowstron. "Past: Persistent and anonymous storage in a peer-to-peer networking environment". In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS), 2001

[63] Circle, http://savannah.nongnu.org/projects/circle/, 2011

[64] J. Ousterhout, et al. “The case for RAMclouds: Scalable high-performance storage entirely in DRAM”. In Operating system review, 2009

[65] J. Dean, S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters,” OSDI 2004

[66] A.L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, R. Schwartzkopf, “The Replica Location Service”, IEEE HPDC, 2004

[67] A. Chervenak, R. Schuler. “The Data Replication Service”, Technical Report, USC ISI, 2006

[68] K. Ranganathan I. Foster, “Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids”, Journal of Grid Computing, V1(1) 2003

[69] T. Kosar. “A New Paradigm in Data Intensive Computing: Stork and the Data-Aware Schedulers,” IEEE CLADE 2006

[70] D.E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, F. Wong. “Parallel computing on the berkeley now”, Symposium on Parallel Processing, 1997

[71] R. Arpaci-Dusseau. “Run-time adaptation in river”, ACM Transactions on Computer Systems, 21(1):36–86, 2003

[72] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, M. Wilde. “Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments”, Grid Computing Research Progress, Nova Publisher 2008

[73] M. Branco, “DonQuijote - Data Management for the ATLAS Automatic Production System”, Computing in High Energy and Nuclear Physics (CHEP04), 2004

[74] D.L. Adams, K. Harrison, C.L. Tan. “DIAL: Distributed Interactive Analysis of Large Datasets”, Computing in High Energy and Nuclear Physics (CHEP 06), 2006

[75] A. Chervenak, et al. “High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies”, Parallel Computing, Special issue on High performance computing with geographical data, 2003

[76] M. Beynon, T.M. Kurc, U.V. Catalyurek, C. Chang, A. Sussman, J.H. Saltz. “Distributed Processing of Very Large Datasets with DataCutter”, Parallel Computing, Vol. 27, No. 11, pp. 1457-1478, 2001

[77] D.T. Liu, M.J. Franklin. “The Design of GridDB: A Data-Centric Overlay for the Scientific Grid”, VLDB04, pp. 600-611, 2004

[78] H. Andrade, T. Kurc, A. Sussman, J. Saltz. "Active Semantic Caching to Optimize Multidimensional Data Analysis in Parallel and Distributed Environments", Parallel Computing Journal, 2007