RC24973 (W1004-026) April 9, 2010

IBM Research Report

PIMD: Parallel In-Memory Database Reference Manual

API version 0.1

Alex Rayshubskiy, Blake Fitch, Mike Pitman, Robert Germain
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

PIMD: Parallel In-Memory Database

Reference Manual

API version 0.1

Alex Rayshubskiy
arayshu@us.ibm.com

Blake Fitch
bgf@us.ibm.com

Mike Pitman
pitman@us.ibm.com

Robert Germain
rgermain@us.ibm.com

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

March 26, 2010

Abstract

Parallel In-Memory Database (PIMD) is a key-value, client-server database system designed to exploit massively parallel, processor-in-memory environments, such as the IBM Blue Gene (BG) family of machines. It provides a set of synchronous and asynchronous data access interfaces, iterators, as well as the ability to restart from a checkpoint. This document describes the technical features, a C++ client API of PIMD, and examples of usage.

Contents

Chapter 1

Parallel In Memory Database Functional Requirements

Target machine characteristics are roughly:

- 10K to 100K nodes – each node to have an insignificant fraction of total memory/cpu/network
- system-on-a-chip processor-in-memory type architecture to achieve scalable, low power, high density
- network focused on High Performance Computing (HPC) – reliable, blocking, limited hardware support for virtualization
- limited external IO compared to internal memory bandwidth and network capability

PIMD Functional Requirements

- persistent (between jobs) parallel multi-ported key-value data sets in main or storage class memory
- internal (within parallel machine) and external (network host) access via a client library
- enable concurrent access from multiple clients
- rebalancing of data distribution transparent to internal/external clients
- respect data access permissions of host environment such as Unix uid/gid
- locking on a record basis
- resiliency against data loss during node failure

Chapter 2

Overview

The design of the Parallel In-Memory Database (PIMD) is influenced by Berkeley DB (BDB) [7] key-value interface and the Remote Direct Memory Access (RDMA) communication paradigm [1].

PIMD is a parallel client, parallel server, key-value database system with basic function similar to BDB. PIMD storage is currently drawn from main memory of the node of the parallel machine. In the future we anticipate PIMD storage will be backed by storage class memory.[5]

Each node contributing storage (memory) runs a parallel PIMD server and collectively are known as a PIMD Server Group. The PIMD client library runs in an MPI parallel environment. It provides an API and manages the client end of the message protocol between the client program and the PIMD server port. PIMD clients and PIMD servers operate on a request - response model. Any valid PIMD command can be sent to any PIMD server. The PIMD server group expects to run on a homogeneous parallel machine. The clients using the PIMD library accessing a PIMD server group may run in the server partition or externally on the host. There is exactly 1 server running on each node of a machine allocation.

The PIMD library provides a standard C/C++ API similar to BDB. Rather than store data to a file, the PIMD library uses a messaging interface to exchange data with the PIMD server group. The PIMD client-server protocol involves short control messages handled in an RPC style exchange and bulk transfers handled via an RDMA type exchange. Client connection to the PIMD server group is managed so that a single connection point provides RDMA access to all servers across the machine.

PIMD is a major component of the Active Storage Fabrics (ASF) project. The goal of ASF is to extend the application reach of the BG family beyond classical HPC workloads. The ASF approach uses the main memory of a massively parallel machine in place of external storage, such as disk, and motivates a re-evaluation of the classic data/storage interface. ASF uses a PIMD server group to manage a significant fraction of the main memory in a distributed memory parallel machine (or partition). PIMD is used to store application or middleware data which enables the application or middleware to transparently offload selected modules into the

parallel environment. Early results of the ASF project have focused on accelerating selected utilities and operations of IBM General Parallel File System (GPFS) and IBM DB2^{TM}. [4]

2.1 The Partitioned Data Set

PIMD uses Partitioned Data Set (PDS) to store tables.

The PDS is the primary structure used to organize PIMD data in the memory of a PIMD server group. A PDS is a distributed container of key-value records. When a PDS is opened a data structure is returned which may be shared with other processes via message passing, broadcast for example, to enable a parallel process to rapidly gain global access.

The PIMD Server that creates a PDS is the PDS *root node*. Each PIMD Server uses a unique ID to name the created PDS. The *unique id* is formed in such a way that the PIMD Server address is an accessible component of the Id. It is up to the creator of the PDS to keep track of the *pds id* including any mapping from a name space map.

A PDS is read and written on a record basis. To access a record, a key must be provided. Parts of records may be read or written. Records may be compressed. Currently, records are distributed to nodes based on a hash of the key, then stored in a red-black tree in memory. Every key hashes to exactly 1 node. A PDS corresponds to a database *table* and a record corresponds to a database *row*.

PIMD client has the ability to request the server group to take a current snapshot of the data image and save it to persistent storage on disk at a specified location. The server group could then be restarted with the data from a snapshot image.

2.1.1 Partitioning/Mapping Scheme

Since PIMD's design goal is to scale to tens of thousands of nodes, the distribution of data amongst the partitions of a PDS is a matter of some concern.

The possibilities for distributing the data include:

- striping
- randomization
- sorting/histogramming and explicit layout.
- hashing

Striping has the disadvantage of requiring a global search to retrieve a value associated with a given key. Its advantages are a layout that's not sensitive to data distribution as well as a simple mechanism for choosing a target node during inserts.

Simply randomizing the primary key to p slots, where p is the number of partitions, will not be sufficient because the distribution will not be perfectly even and therefore one node's storage may be exhausted while the system as a whole is nowhere near full. This is particularly problematic for large ratios of m/p, where m is the number of records. The determination of the maximum occupancy of a partition for values of $m/p \gg 1$ has been solved[2] and the expression for the "maximum" occupancy (at least the "maximum" with probability of $O(1)$) in the limit where $m \gg p\,(\log p)^3$ is

$$\frac{m}{p} + \sqrt{\frac{2\,m \log p}{p}\left(1 - \frac{\log^2 p}{2\,\alpha\,\log p}\right)} \tag{2.1}$$

where $0 < \alpha < 1$.

A sorting/histogramming approach with bin boundaries determined by the histogram could work, but there are a couple of issues to consider:

- The initial distribution could be extremely non-uniform so that even an initial load of the data might necessitate remapping during the load.
- If data is added subsequent to the initial load (or if the data is loaded incrementally), then the distribution may change dramatically necessitating frequent re-histogramming and distribution operations.

The current implementation of PIMD uses hashing to distribute data to the servers. A hash function from a universal class has been chosen due to its simplicity, favorable collision properties and speed of computation.[3]

2.2 Current execution environment of PIMD

- Linux kernel: 2.6.29
- MPI: MPICH2 1.0.8
- RDMA verbs: Open source Soft iWARP implementation from the IBM Zurich Research Lab in the OFED environment[1, 6]

2.3 Future Work

- Export / import per PDS
- Load balancing
- Resiliency

Chapter 3

PIMD C++ Client Library API Reference

3.1 Declaration

pimd_client_t — class defining all pimd client methods

#include <pimd_client.hpp>

All PIMD client methods return a status code (pimd_status_t). See below.

```
void pimd_client_t();
```

3.2 Init

```
pimd_status_t Init( pimd_client_group_id_t  aCommGroupId,
                    MPI_Comm                aComm,
                    int                     aFlags );

pimd_status_t Finalize();
```

Initializes\Finalizes client's resources
aCommGroupId - intenger identifier of the pimd client group in the process. There could be several client groups in the same process.
aComm - MPI communicator representing the tasks composing the client group. Often this is MPI_COMM_WORLD.
aFlags - This is a place holder for future initialization flags. For now this parameter is not active.

3.3 Connection to PIMD server

```
pimd_status_t Connect( char* aServerGroupName, int aFlags );
pimd_status_t Disconnect();
```

aServerGroupName - Specifies a name of the server group. When targeting PIMD servers running on BlueGene/P, this is the block id of the partition where the server group is running.

aFlags - This is a place holder for future connection flags. For now this parameter is not active.

3.4 Synchronous Commands

PIMD provides a set of synchronous interfaces to open PDSs and access data. Each synchronous method will block until the desired action is completed.

3.4.1 Open a PSD

Opening a PDS is similar to opening a file handle via the POSIX open() function.

```
pimd_status_t Open( char*                   aPDSName,
                    pimd_pds_priv_t         aPrivs,
                    pimd_cmd_open_flags_t   aFlags,
                    pimd_pds_id_t*          aPDSId );
```

aPDSName - Null-terminated name of the PDS to create or open depending on the *aFlags*. Current limit 64 characters.

aPrivs - Open privileges for the PDS, with semantics similar to POSIX. Current privileges are available below.

```
typedef enum
  {
    PIMD_PDS_READ                       = 0x0001,
    PIMD_PDS_WRITE                      = 0x0002
  } pimd_pds_priv_t;
```

aFlags - Flags specify if an existing PDS should be opened or if a new PDS should be created

```
typedef enum
  {
    PIMD_COMMAND_OPEN_FLAGS_NONE        = 0x0000,

    // Create if does not exist
    PIMD_COMMAND_OPEN_FLAGS_CREATE      = 0x0001,
```

```
    // Return error if already exists
    PIMD_COMMAND_OPEN_FLAGS_EXCLUSIVE   = 0x0002,

    // PDS has duplicates
    PIMD_COMMAND_OPEN_FLAGS_DUP         = 0x0003

  } pimd_cmd_open_flags_t;
```

PIMD_COMMAND_OPEN_FLAGS_NONE - try to open a PDS, return an error if does not exist.
PIMD_COMMAND_OPEN_FLAGS_CREATE - try to open a PDS, create if does not.
PIMD_COMMAND_OPEN_FLAGS_EXCLUSIVE - try to create a PDS, return an error if exist.

NOTE: PIMD_COMMAND_OPEN_FLAGS_DUP is currently not supported.

aPDSId - Handle to the opened PDS structure. The structure should be allocated by the caller.

Return: PIMD_SUCCESS if the PDS was successfully opened. Otherwise, a PIMD errno. Possible list is below:

```
PIMD_ERRNO_PDS_ALREADY_EXISTS
PIMD_ERRNO_PDS_DOES_NOT_EXIST
PIMD_ERRNO_PERMISSION_DENIED
```

3.4.2 Insert a record

```
  pimd_status_t Insert( pimd_pds_id_t*        aPDSId,
                        char*                 aKeyBuffer,
                        int                   aKeyBufferSize,
                        char*                 aValueBuffer,
                        int                   aValueBufferSize,
                        int                   aValueBufferOffset,
                        pimd_cmd_RIU_flags_t  aFlags );
```

aPDSId - PDS handle as returned by the Open() call.
aKeyBuffer - pointer to the key data.
aKeyBufferSize - size of the key data.
aValueBuffer - pointer to the value data buffer.
aValueBufferSize - size of value data buffer.
aValueBufferOffset - Offset in value portion of the record to insert the data into.
aFlags - This is a bit mask of flags chosen from the Remove Update Insert (RUI) flags below.

```
typedef enum
  {
```

```
    PIMD_COMMAND_RIU_FLAGS_NONE                 = 0x0000,

    PIMD_COMMAND_RIU_INSERT_EXPANDS_VALUE       = 0x0001,

    PIMD_COMMAND_RIU_USE_RECORD_LOCKS           = 0x0002,

  } pimd_cmd_RIU_flags_t;
```

PIMD_COMMAND_RIU_INSERT_EXPANDS_VALUE - This flag is only valid for the insert command. If this flag is specified, the insert will create a buffer equal to Offset+Len, if necessary NOTE: Every time a new buffer is created a memcpy of the already present data is performed

PIMD_COMMAND_RIU_USE_RECORD_LOCKS - Turns on record locking. Useful if clients wish to access the same record concurrently.

Return: PIMD_SUCCESS if the PDS was successfully opened. Otherwise, a PIMD errno. Possible list is below:

```
PIMD_ERRNO_RECORD_ALREADY_EXISTS
PIMD_ERRNO_NODE_FULL
PIMD_ERRNO_KEY_TOO_LARGE
PIMD_ERRNO_VALUE_TOO_LARGE
```

3.4.3 Retrieve a record

```
  pimd_status_t Retrieve( pimd_pds_id_t*          aPDSId,
                          char*                   aKeyBuffer,
                          int                     aKeyBufferSize,
                          char*                   aValueBuffer,
                          int                     aValueBufferSize,
                          int*                    aValueRetrievedSize,
                          int                     aOffset,
                          pimd_cmd_RIU_flags_t    aFlags );
```

aPDSId - PDS handle as returned by the Open() call.
aKeyBuffer - pointer to the key data.
aKeyBufferSize - size of the key data.
aValueBuffer - pointer to the output value data buffer.
aValueBufferSize - max size of output value data buffer.
aValueRetrievedSize - actual retrieved value data size.
aOffset - Offset in value portion of the record to retrieve the data from.
aFlags - This is a bit mask of flags chosen from the RUI flags above.

Return: PIMD_SUCCESS if the PDS was successfully retrieved. Otherwise, a PIMD errno. Possible list is below:

PIMD_ERRNO_KEY_NOT_FOUND PIMD_ERRNO_NODE_FULL PIMD_ERRNO_KEY_TOO_LARGE PIMD_ERRNO_VALUE_TOO_LARGE

3.4.4 Update a record

```
pimd_status_t Update( pimd_pds_id_t*          aPDSId,
                      char*                   aKeyBuffer,
                      int                     aKeyBufferSize,
                      char*                   aValueBuffer,
                      int                     aValueUpdateSize,
                      int                     aOffset,
                      pimd_cmd_RIU_flags_t    aFlags );
```

aPDSId - PDS handle as returned by the Open() call.
aKeyBuffer - pointer to the key data.
aKeyBufferSize - size of the key data.
aValueBuffer - pointer to the value data buffer.
aValueUpdateSize - size of value data to update.
aOffset - Offset in value portion of the record to update the data into.
aFlags - This is a bit mask of flags chosen from the RUI flags above.

Return: PIMD_SUCCESS if the PDS was successfully updated. Otherwise, a PIMD errno. Possible list is below:

PIMD_ERRNO_KEY_NOT_FOUND PIMD_ERRNO_NODE_FULL PIMD_ERRNO_KEY_TOO_LARGE PIMD_ERRNO_VALUE_TOO_LARGE

3.4.5 Remove a record

```
pimd_status_t Remove( pimd_pds_id_t*          aPDSId,
                      char*                   aKeyBuffer,
                      int                     aKeyBufferSize,
                      pimd_cmd_RIU_flags_t    aFlags );
```

aPDSId - PDS handle as returned by the Open() call.
aKeyBuffer - pointer to the key data.

aKeyBufferSize - size of the key data.
aFlags - This is a bit mask of flags chosen from the RUI flags above.

Return: PIMD_SUCCESS if the PDS was successfully removed. Otherwise, a PIMD errno. Possible list is below:

PIMD_ERRNO_KEY_NOT_FOUND PIMD_ERRNO_KEY_TOO_LARGE

3.4.6 Close a PDS

```
pimd_status_t Close( pimd_pds_id_t* aPDSId );
```

aPDSId - PDS handle as returned by the Open() call. After the Close() the handle will no longer be valid.

3.4.7 Synchronous Commands Example

```
#include <pimd_client.hpp>
#include <FxLogger.hpp>

pimd_client_t Client;

#define DATA_SIZE         ( 4096 )

int
main(int argc, char **argv)
{
  printf( "pimd_client::entering main \n" ); fflush( stdout );

  char* ServerAddress = argv[ 1 ];

  BegLogLine( 1 )
    << "TestClientDriver::main():: About to connect "
    << " ServerAddress: " << ServerAddress
    << EndLogLine;

  /*************************************************************************
   * Init MPI
   *************************************************************************/
  MPI_Init( &argc, &argv );
  int Rank;
  int NodeCount;
  MPI_Comm_rank( MPI_COMM_WORLD, &Rank );
  MPI_Comm_size( MPI_COMM_WORLD, &NodeCount );
  /*************************************************************************/



  /*************************************************************************
   * Init the PIMD Client
   *************************************************************************/
  pimd_status_t status = Client.Init( 0, MPI_COMM_WORLD, 0 );

  StrongAssertLogLine( status == PIMD_SUCCESS )
    << " status: " << pimd_status_to_string( status )
    << EndLogLine;
  /*************************************************************************/



  /*************************************************************************
   * Connect to the PIMD Server
   *************************************************************************/
  status = Client.Connect( ServerAddress, 0 );
  StrongAssertLogLine( status == PIMD_SUCCESS )
    << " status: " << pimd_status_to_string( status )
    << EndLogLine;
  /*************************************************************************/
```

```
/*****************************************************************************
 * Open a test PDS
 *****************************************************************************/
pimd_pds_id_t  MyPDSId;

if( Rank == 0 )
  {
    status = Client.Open("MyPDS",
                         (pimd_pds_priv_t)(PIMD_PDS_READ | PIMD_PDS_WRITE),
                         (pimd_cmd_open_flags_t)PIMD_COMMAND_OPEN_FLAGS_CREATE,
                         & MyPDSId );

    StrongAssertLogLine( status == PIMD_SUCCESS )
      << " status: " << pimd_status_to_string( status )
      << EndLogLine;
  }

MPI_Bcast( (char *) & MyPDSId,
           sizeof( pimd_pds_id_t ),
           MPI_CHAR,
           0,
           MPI_COMM_WORLD );
/*****************************************************************************/



/*****************************************************************************
 * Allocate Insert/Retrieve data arrays
 *****************************************************************************/
// Create Insert Key / Value
int    MyDataInsertSize = DATA_SIZE;
char* MyDataInsert    = (char *) malloc( MyDataInsertSize );
StrongAssertLogLine( MyDataInsert != NULL )
  << "TestClientDriver::main():: ERROR:: MyDataInsert != NULL"
  << EndLogLine;

// Allocate Retrieve Data
char* MyDataRetrieve    = (char *) malloc( MyDataInsertSize );
StrongAssertLogLine( MyDataRetrieve != NULL )
  << "TestClientDriver::main():: ERROR:: MyDataRetrieve != NULL"
  << EndLogLine;


int TestFailed = 0;
for( int t=0; t < NUMBER_OF_TRIES; t++ )
  {
    /*************************************************************
     * Insert Key / Value
     *************************************************************/
    for( int i=0; i < MyDataInsertSize; i++ )
      {
        char ch = i;
        MyDataInsert[ i ] = ch;
      }
```

```
int Key = t;

BegLogLine( 1 )
  << "TestClientDriver::main():: About to Insert "
  << " into MyPDSId: " << MyPDSId
  << EndLogLine;

status = Client.Insert( &MyPDSId,
                        (char *) &Key,
                        sizeof( int ),
                        MyDataInsert,
                        MyDataInsertSize,
                        0,
                        PIMD_COMMAND_RIU_FLAGS_NONE );

StrongAssertLogLine( status == PIMD_SUCCESS )
  << " status: " << pimd_status_to_string( status )
  << EndLogLine;
/*********************************************************************/


/*********************************************************************
 * Retrieve Key / Value
 *********************************************************************/
bzero( MyDataRetrieve, MyDataInsertSize );

int RetrieveSize = 0;
status = Client.Retrieve( &MyPDSId,
                          (char *) &Key,
                          sizeof( int ),
                          MyDataRetrieve,
                          MyDataInsertSize,
                          & RetrieveSize,
                          0,
                          PIMD_COMMAND_RIU_FLAGS_NONE );

StrongAssertLogLine( status == PIMD_SUCCESS )
  << " status: " << pimd_status_to_string( status )
  << EndLogLine;
/*********************************************************************/



/*********************************************************************
 * Check results
 *********************************************************************/
int Match = 1;
for( int i=0; i<MyDataInsertSize; i++ )
  {
    char ch = i;
    if( MyDataRetrieve[ i ] != ch )
      {
        BegLogLine( 1 )
          << "TestClientDriver::main():: Retrieve Result mismatch: {"
```

```
                << MyDataRetrieve[ i ] << " , "
                << ch << " }"
                << EndLogLine;

              Match = 0;
            }
        }

      if( !Match )
        {
          TestFailed = 1;
          break;
        }
      /*****************************************************************/
    }

  if( TestFailed )
    {
      BegLogLine( 1 )
        << "TestClientDriver::main():: PIMD Client Result Match FAILED :-("
        << EndLogLine;
    }
  else
    {
      BegLogLine( 1 )
        << "TestClientDriver::main():: PIMD Client Result Match SUCCESSFUL :-)"
        << EndLogLine;
    }

  return 0;
}
```

3.5 Asynchronous Commands

PIMD provides a set of asynchronous (non-blocking) interfaces to open PDSs and access data. Each asynchronous method will return a handle - pimd_client_cmd_ext_hdl_t* - to the PIMD command. Semantics for the Open(), Insert(), Update(), Retrieve() Remove() are exactly the same as their synchronous counterparts.

Poll for command completion of a command via Test() or block waiting on the handle via Wait().

Poll for completion of **any** command via TestAny() or block waiting on **any** command completion via WaitAny().

This is a useful feature if the application doesn't want to pay the latency cost of every command. The asynchronous interface allows the application to have multiple outstanding commands, and a higher throughput of data transfer.

```
pimd_status_t iOpen( char*                    aPDSName,
                     pimd_pds_priv_t          aPrivs,
                     pimd_cmd_open_flags_t    aFlags,
                     pimd_pds_id_t*           aPDSId,
                     pimd_client_cmd_ext_hdl_t* aCmdHdl );

pimd_status_t iRetrieve( pimd_pds_id_t*           aPDSId,
                         char*                    aKeyBuffer,
                         int                      aKeyBufferSize,
                         char*                    aValueBuffer,
                         int                      aValueBufferSize,
                         int*                     aValueRetrievedSize,
                         int                      aOffset,
                         pimd_cmd_RIU_flags_t     aFlags,
                         pimd_client_cmd_ext_hdl_t*  aCmdHdl );

pimd_status_t iUpdate( pimd_pds_id_t*           aPDSId,
                       char*                    aKeyBuffer,
                       int                      aKeyBufferSize,
                       char*                    aValueBuffer,
                       int                      aValueUpdateSize,
                       int                      aOffset,
                       pimd_cmd_RIU_flags_t     aFlags,
                       pimd_client_cmd_ext_hdl_t*  aCmdHdl );

pimd_status_t iInsert( pimd_pds_id_t*           aPDSId,
                       char*                    aKeyBuffer,
                       int                      aKeyBufferSize,
                       char*                    aValueBuffer,
                       int                      aValueBufferSize,
                       int                      aValueBufferOffset,
                       pimd_cmd_RIU_flags_t     aFlags,
```

```
                        pimd_client_cmd_ext_hdl_t*  aCmdHdl );

pimd_status_t iRemove( pimd_pds_id_t*          aPDSId,
                       char*                   aKeyBuffer,
                       int                     aKeyBufferSize,
                       pimd_cmd_RIU_flags_t    aFlags,
                       pimd_client_cmd_ext_hdl_t*  aCmdHdl );

pimd_status_t iClose( pimd_pds_id_t*           aPDSId,
                      pimd_client_cmd_ext_hdl_t*   aCmdHdl );

// Wait

// Returns a handle that has completed
pimd_status_t WaitAny( pimd_client_cmd_ext_hdl_t* aCmdHdl );

pimd_status_t Wait( pimd_client_cmd_ext_hdl_t aCmdHdl );

// Test

// Returns a completed handle on success
pimd_status_t TestAny( pimd_client_cmd_ext_hdl_t* aCmdHdl );

pimd_status_t Test( pimd_client_cmd_ext_hdl_t aCmdHdl );
```

3.5.1 Asynchronous Commands Example

```
#include <pimd_client.hpp>
#include <FxLogger.hpp>

pimd_client_t Client;

#define DATA_SIZE            ( 4096 )
#define DATA_ELEMENT_COUNT   ( 256 )

struct data_elem_t
{
  char data[ DATA_SIZE ];
};

data_elem_t                 gInsertDataElements[ DATA_ELEMENT_COUNT ];
pimd_client_cmd_ext_hdl_t   gInsertCommandHandleSet[ DATA_ELEMENT_COUNT ];

linked_list_t               gLinkedListOfCommandHandles;

int
main(int argc, char **argv)
{

  ... Init(), Connect(), Open() as in the syncronous example

  /*************************************************************************
   * Insert data into PIMD asynchronously
   *************************************************************************/
  for( int t = 0; t < DATA_ELEMENT_COUNT; t++ )
    {
      /*********************************************************************
       * Insert Key / Value
       *********************************************************************/
      for( int i = 0; i < DATA_SIZE; i++ )
        {
          char ch = i;
          gInsertDataElements[ t ].data[ i ] = ch;
        }

      int Key = Rank * DATA_ELEMENT_COUNT + t;

      gCompletedFlag[ t ] = 0;

      BegLogLine( 1 )
        << "TestClientDriver::main()::  About to Insert "
        << " into MyPDSId: " << MyPDSId
        << EndLogLine;

      status = Client.iInsert( &MyPDSId,
                               (char *) &Key,
                               sizeof( int ),
                               gInsertDataElements[ t ].data,
                               DATA_SIZE,
                               0,
```

```
                                    PIMD_COMMAND_RIU_FLAGS_NONE,
                                    & gInsertCommandHandleSet[ t ] );

      gLinkedListOfCommandHandles.Insert( & gInsertCommandHandleSet[ t ] );

      StrongAssertLogLine( status == PIMD_SUCCESS )
        << " status: " << pimd_status_to_string( status )
        << EndLogLine;
      /*****************************************************************/
    }

  /*************************************************************************
   * Wait on the insert to complete
   ************************************************************************/
#ifdef WAIT_ANY
  for( int t = 0; t < DATA_ELEMENT_COUNT; t++ )
    {
      pimd_client_cmd_ext_hdl_t completedHandle;
      Client.WaitAny( & completedHandle );
    }
#elif WAIT
  for( int t = 0; t < DATA_ELEMENT_COUNT; t++ )
    {
      Client.Wait( & gInsertCommandHandleSet[ t ] );
    }
#elif TEST_ANY
  int CompletedCount = 0;
  while( CompletedCount < DATA_ELEMENT_COUNT )
    {
      pimd_client_cmd_ext_hdl_t completedHandle;
      pimd_status_t status = Client.TestAny( & completedHandle );
      if( status == PIMD_SUCCESS )
        {
          CompletedCount++;
        }
    }
#elif TEST

  link_node_t* currentNode = gLinkedListOfCommandHandles.head();
  link_node_t* nextNode = NULL;

  // Iterate over the linked list until there
  // are no more elements.
  while( gLinkedListOfCommandHandles.size() > 0 )
    {
      nextNode = currentNode->next;

      pimd_status_t status = Client.Test( currentNode->elem );
      if( status == PIMD_SUCCESS )
        {
          gLinkedListOfCommandHandles.Remove( currentNode );
        }

      if(nextNode == NULL)
        currentNode = gLinkedListOfCommandHandles.head();
```

```
      else
        currentNode = nextNode;
    }
#endif
  /*************************************************************************/

  return 0;
}
```

3.6 Iterators

PIMD provides interfaces for iterating over the records of a PDS. There are 2 kinds of iterators:

- Local iterators - Retrieve records of a PDS from a single node.
- Global iterators - Retrieve all records of a PDS.

3.6.1 Local Iterators

```
pimd_status_t OpenLocalCursor(
    int                              aNodeId,
    pimd_pds_id_t*                   aPDSId,
    pimd_client_cursor_ext_hdl_t*    aCursorHdl );

pimd_status_t CloseLocalCursor(
    pimd_client_cursor_ext_hdl_t     aCursorHdl );
```

aNodeId - MPI rank of the server node to which an iterator will be opened.
aPDSId - PDS handle as returned by the Open() call.
aCursorHdl - Handle to the opened local cursor.

Return: PIMD_SUCCESS if the local cursor was successfully opened. Otherwise, a PIMD errno. Possible list is below:

```
PIMD_ERRNO_PDS_DOES_NOT_EXIST
PIMD_ERRNO_INVALID_CURSOR_RANK
```

NOTE: The cursor nomenclature somes from BDB.

```
pimd_status_t GetFirstLocalElement(
    pimd_client_cursor_ext_hdl_t aCursorHdl,
    char*                        aRetrievedKeyBuffer,
    int*                         aRetrievedKeySize,
    int                          aRetrievedKeyMaxSize,
    char*                        aRetrievedValueBuffer,
    int*                         aRetrievedValueSize,
    int                          aRetrievedValueMaxSize,
    pimd_cursor_flags_t          aFlags );

pimd_status_t GetNextLocalElement(
    pimd_client_cursor_ext_hdl_t  aCursorHdl,
    char*                         aRetrievedKeyBuffer,
    int*                          aRetrievedKeySize,
    int                           aRetrievedKeyMaxSize,
    char*                         aRetrievedValueBuffer,
    int*                          aRetrievedValueSize,
```

```
    int                         aRetrievedValueMaxSize,
    pimd_cursor_flags_t         aFlags );
```

aCursorHdl - Handle to the opened local cursor as returned by the OpenLocalCursor() call.
aRetrievedKeyBuffer - pointer to the buffer that will contain a retrieved key.
aRetrievedKeySize - pointer to the returned retrieved key size.
aRetrievedKeyMaxSize - maximum size of the retrieved key buffer.
aRetrievedValueBuffer - pointer to the buffer that will contain a retrieved value.
aRetrievedValueSize - pointer to the returned retrieved value size.
aRetrievedValueMaxSize - maximum size of the retrieved value buffer.
aFlags - Flags are listed below:

```
typedef enum
  {
    PIMD_CURSOR_NONE_FLAG,
    PIMD_CURSOR_WITH_STARTING_KEY_FLAG,
  } pimd_cursor_flags_t;
```

PIMD_CURSOR_WITH_STARTING_KEY_FLAG - Specifies for the first element retrieved to be $\geq$ then the key specified in *aRetrievedKeyBuffer*, *aRetrievedKeySize*

Return: PIMD_SUCCESS if a record was successfully retrieved. PIMD_ERRNO_END_OF_RECORDS if there are no more records to retrieve. Otherwise, a PIMD errno. Possible list is below:

```
PIMD_ERRNO_RETRIEVE_BUFFER_MAX_SIZE_EXCEEDED
PIMD_ERRNO_IT_POST_SEND_FAILED
PIMD_ERRNO_IT_POST_RECV_FAILED
PIMD_ERRNO_IT_POST_RDMA_WRITE_FAILED
```

3.6.2 Global Iterators

Global iterator's interface and semantics are almost identical to the local iterator. This iterator will retrieve all the records associated with a specified PDS: *aPDSId*

The differences are:

- Method names don't have "Local" in them.
- OpenCursor() method does not take a node id as an argument.

```
  pimd_status_t OpenCursor( pimd_pds_id_t*                aPDSId,
                            pimd_client_cursor_ext_hdl_t* aCursorHdl );

  pimd_status_t CloseCursor( pimd_client_cursor_ext_hdl_t aCursorHdl );

  pimd_status_t GetFirstElement(
```

```
    pimd_client_cursor_ext_hdl_t aCursorHdl,
    char*                        aRetrievedKeyBuffer,
    int*                         aRetrievedKeySize,
    int                          aRetrievedKeyMaxSize,
    char*                        aRetrievedValueBuffer,
    int*                         aRetrievedValueSize,
    int                          aRetrievedValueMaxSize,
    pimd_cursor_flags_t          aFlags );

pimd_status_t GetNextElement(
    pimd_client_cursor_ext_hdl_t  aCursorHdl,
    char*                         aRetrievedKeyBuffer,
    int*                          aRetrievedKeySize,
    int                           aRetrievedKeyMaxSize,
    char*                         aRetrievedValueBuffer,
    int*                          aRetrievedValueSize,
    int                           aRetrievedValueMaxSize,
    pimd_cursor_flags_t           aFlags );
```

3.6.3 Iterators Example

```
pimd_status_t pstatus = aPIMDClient->OpenLocalCursor( MyNodeId,
                                                      aBlocksPDSId,
                                                      & LocalBlocksCursors );

StrongAssertLogLine( pstatus == PIMD_SUCCESS )
  << "uu_blocks_to_records():: ERROR:: "
  << " status: " << pimd_status_to_string( pstatus )
  << EndLogLine;

uu_gpfs_key_t KeyBuffer;
KeyBuffer.Init( aFileInode, 0 );
bzero( & KeyBuffer, sizeof( uu_gpfs_key_t ) );

int KeySize = sizeof( uu_gpfs_key_t );
int RetrievedValueSize = -1;
int MaxValueSize = UU_GPFS_BLOCK_SIZE;

pimd_cursor_flags_t CursorFlags = PIMD_CURSOR_WITH_STARTING_KEY_FLAG;

pstatus = aPIMDClient->GetFirstLocalElement( LocalBlocksCursors,
                                             (char *) & KeyBuffer,
                                             & KeySize,
                                             KeySize,
                                             BlockDataBuffer,
                                             & RetrievedValueSize,
                                             MaxValueSize,
                                             CursorFlags );

BegLogLine( UU_BLOCK_TO_RECORDS_LOG )
  << "uu_block_to_records():: After GetFirstLocalElement():: "
  << " pstatus: " << pimd_status_to_string( pstatus )
  << EndLogLine;

CursorFlags = (pimd_cursor_flags_t) 0;

while( pstatus != PIMD_ERRNO_END_OF_RECORDS )
  {

    // Process the key / value pair
    ...

    pstatus = aPIMDClient->GetNextLocalElement( LocalBlocksCursors,
                                                (char *) & KeyBuffer,
                                                & KeySize,
                                                KeySize,
                                                BlockDataBuffer,
                                                & RetrievedValueSize,
                                                MaxValueSize,
                                                CursorFlags );
  }

pstatus = aPIMDClient->CloseLocalCursor( LocalBlocksCursors );
StrongAssertLogLine( pstatus != PIMD_SUCCESS )
```

```
    << EndLogLine;
```

3.7 Bulk Inserts

Since inserting a large number of records into a database is a common operation, PIMD provides an optimized set of interfaces for executing bulk inserts. A number of records are buffered on the client and are only transferred to the server when the buffer gets full. By transferring one large buffer instead of many small ones, PIMD can cut down on message processing overhead and take advantage of the large message transfer benefits of the RDMA.

If the application needs to ensure that certain data gets to the server, the Flush() call is available. Flush() also gets called by CloseBulkInserter().

```
pimd_status_t CreateBulkInserter(
    pimd_pds_id_t*                          aPDSId,
    pimd_bulk_inserter_flags_t              aFlags,
    pimd_client_bulk_inserter_ext_hdl_t*    aBulkInserterHandle );
```

aPDSId - PDS handle as returned by the Open() call.
aFlags - This is a placeholder for future bulk inserter flags.
aBulkInserterHandle - returned handle to an initialized bulk inserter.

```
pimd_status_t Insert(
    pimd_client_bulk_inserter_ext_hdl_t   aBulkInserterHandle,
    char*                                 aKeyBuffer,
    int                                   aKeyBufferSize,
    char*                                 aValueBuffer,
    int                                   aValueBufferSize,
    pimd_bulk_inserter_flags_t            aFlags );

pimd_status_t Flush(
    pimd_client_bulk_inserter_ext_hdl_t  aBulkInserterHandle );

pimd_status_t CloseBulkInserter(
     pimd_client_bulk_inserter_ext_hdl_t aBulkInserterHandle );
```

aBulkInserterHandle - returned handle to an initialized bulk inserter.
aKeyBuffer - buffer holding key data.
aKeyBufferSize - size of the key.
aValueBuffer - buffer holding value data.
aValueBufferSize - size of the value.
aFlags - This is a placeholder for future bulk inserter flags.

Return: PIMD_SUCCESS if a record was successfully inserted. Otherwise, a PIMD errno.

3.7.1 Bulk Inserts Example

This example reads in an input file with data that can be decomposed into { key , value } and bulk inserts records into PIMD.

```
/*****************************************************************************
 * Init the PIMD Client
 *****************************************************************************/
pimd_status_t status = gPimdClient.Init( 0, MPI_COMM_WORLD, 0 );

if( status == PIMD_SUCCESS )
  {
    BegLogLine( 1 )
      << "pimd_file_loader::main():: PIMD Client Init succeded "
      << EndLogLine;
  }
else
  {
    BegLogLine( 1 )
      << "pimd_file_loader::main():: PIMD Client Init FAILED "
      << " status: " << pimd_status_to_string( status )
      << EndLogLine;
  }
/*****************************************************************************/



/*****************************************************************************
 * Connect to the PIMD Server
 *****************************************************************************/
status = gPimdClient.Connect( BlockName, 0 );

if( status == PIMD_SUCCESS )
  {
    BegLogLine( 1 )
      << "pimd_file_loader::main():: PIMD Client connected "
      << EndLogLine;
  }
else
  {
    BegLogLine( 1 )
      << "pimd_file_loader::main():: PIMD Client FAILED to connect"
      << " status: " << pimd_status_to_string( status )
      << EndLogLine;
  }
/*****************************************************************************/

pimd_pds_priv_t privs = (pimd_pds_priv_t )(PIMD_PDS_READ | PIMD_PDS_WRITE);
pimd_cmd_open_flags_t flags = PIMD_COMMAND_OPEN_FLAGS_CREATE;
pimd_pds_id_t pdsId;
pimd_cmd_RIU_flags_t rflags = PIMD_COMMAND_RIU_FLAGS_NONE;
```

```
  /*****************************************************************************
   * Open the PIMD Server
   *****************************************************************************/
  char name[64];
  sprintf(name,"%s",TsDef);
  BegLogLine( 1 )
    << "pimd_file_loader::main():: Opening pds "
    << name << " Seedname " << TsDef
    << EndLogLine;

  status = gPimdClient.Open( name, privs, flags, &pdsId );

  if( status == PIMD_SUCCESS )
    {
      BegLogLine( 1 )
        << "pimd_file_loader::main():: pds opened "
        << EndLogLine;
    }
  else
    {
      BegLogLine( 1 )
        << "pimd_file_loader:::main():: PIMD Client FAILED to open pds"
        << " status: " << pimd_status_to_string( status )
        << EndLogLine;
    }
  /*****************************************************************************/

#define BUFLEN 100000
  char row_buffer[BUFLEN];
  int KeyLen,ValueLen;
  char* Key = NULL;
  char* Value = NULL;

  // Open input file
  FILE *rowfilePF = fopen64( FName, "r" );

  /*****************************************************************************
   * Create a Bulk Inserter
   *****************************************************************************/
  pimd_client_bulk_inserter_ext_hdl_t BulkLoaderHandle;
  status = gPimdClient.CreateBulkInserter( &pdsId,
                                           (pimd_bulk_inserter_flags_t) 0,
                                           &BulkLoaderHandle );
  AssertLogLine( status == PIMD_SUCCESS )
    << "pimd_file_loader:: ERROR:: "
    << " status: " << pimd_status_to_string( status )
    << EndLogLine;
  /*****************************************************************************/

  int nRecs = 0;
  while(!feof(rowfilePF))
    {
      if(Stash.ReadRow(rowfilePF,row_buffer,&Key, KeyLen,&Value,ValueLen, BUFLEN))
        {
          Stash.SetRowForOutput(row_buffer);
```

```
          BegLogLine( PIMD_FILE_LOADER_LOG )
            << "pimd_file_loader::main():: inserted record "
            << nRecs << " "
            << Stash
            << EndLogLine;

          status = gPimdClient.Insert( BulkLoaderHandle,
                                       Key, KeyLen, Value, ValueLen,
                                       (pimd_bulk_inserter_flags_t) 0 );

          if( status == PIMD_SUCCESS )
            {
              nRecs++;
            }
          else
            {
              BegLogLine( 1 )
                << "pimd_file_loader::main()::PIMD Client FAILED insert record"
                << nRecs
                << " status: " << pimd_status_to_string( status )
                << EndLogLine;
            }

          StrongAssertLogLine( status == PIMD_SUCCESS )
            << "ERROR: "
            << " status: " << pimd_status_to_string( status )
            << EndLogLine;
        }
    }
  fclose(rowfilePF);

  status = gPimdClient.CloseBulkInserter( BulkLoaderHandle );

  AssertLogLine( status == PIMD_SUCCESS )
    << "pimd_file_loader:: ERROR:: "
    << " status: " << pimd_status_to_string( status )
    << EndLogLine;
```

3.8 Persisting snapshots of data

PIMD provides the ability to save an image of the data in the server group to persistent storage (disk). The server can be started with the saved image, by simply passing it a path to the image files that were created during a snapshot. Each server compresses an image of its data and copies it to a path specified by the user.

Current restrictions:

- Snapshot is taken of all the data. We're working on providing the ability to take snapshots on a PDS basis.
- Restart PDS server group must be the same size as the server group which took the snapshot.

```
pimd_status_t DumpPersistentImage( char* aPath );
```

aPath - Path to persistent storage on the host machine which is visible to the nodes running the servers.

3.8.1 Snapshot Example

```
#include <pimd_client.hpp>

void
usage(char* aProgname )
{
  printf( "usage(): %s <block id> <target datapath>\n", aProgname );
}

int
main(int argc, char **argv)
{
  /*************************************************************************
   * Init MPI
   *************************************************************************/
  MPI_Init( &argc, &argv );
  int Rank;
  int NodeCount;
  MPI_Comm_rank( MPI_COMM_WORLD, &Rank );
  MPI_Comm_size( MPI_COMM_WORLD, &NodeCount );
  /*************************************************************************/

  pimd_client_t* PimdClient = (pimd_client_t *) malloc( sizeof( pimd_client_t ));
  assert( PimdClient );

  if( argc != 3 )
    {
      usage( argv[ 0 ] );
      exit( -1 );
    }

  char* BlockName              = argv[ 1 ];
  char* TargetSnapshotDataPath = argv[ 2 ];

  pimd_status_t status = PimdClient->Init( 0, MPI_COMM_WORLD, 0 );
  if( status != PIMD_SUCCESS )
    {
      fprintf(stderr, "ERROR: %s", pimd_status_to_string( status ) );
      fflush(stderr); exit( -1 );
    }

  status = PimdClient->Connect( BlockName, 0 );
  if( status != PIMD_SUCCESS )
    {
      fprintf(stderr, "ERROR: %s", pimd_status_to_string( status ) );
      fflush(stderr); exit( -1 );
    }

  status = PimdClient->DumpPersistentImage( TargetSnapshotDataPath );
  if( status != PIMD_SUCCESS )
    {
      PimdClient->Disconnect();
      fprintf(stderr, "ERROR: %s", pimd_status_to_string( status ) );
      fflush(stderr); exit( -1 );
```

```
    }

  BegLogLine( 1 )
    << "pimd_snapshot: Finished with taking a pimd snapshot to: "
    << TargetSnapshotDataPath
    << EndLogLine;

  PimdClient->Disconnect();
  PimdClient->Finalize();

  return 0;
}
```

3.9 pimd_status_t

#include <pimd_errno.hpp>

```
typedef enum
  {
  // This one is the best!
  PIMD_SUCCESS,

  // Node is not an owner
  PIMD_ERRNO_NODE_NOT_OWNER,

  // Node is full, time to rebalance
  PIMD_ERRNO_NODE_FULL,

  // No record in the database
  PIMD_ERRNO_ELEM_NOT_FOUND,

  // Record already exists in the database
  PIMD_ERRNO_RECORD_ALREADY_EXISTS,

  // The iterator reached the end
  PIMD_ERRNO_ITER_DONE,

  // No local IP addresses were found
  PIMD_ERRNO_LOCAL_IP_NOT_FOUND,

  // Client's connection was refused
  PIMD_ERRNO_CONNECTION_REQUEST_REFUSED,

  // No event was found on an evd
  PIMD_ERRNO_NO_EVENT,

  // Connection failed
  PIMD_ERRNO_CONN_FAILED,

  // Command limit reached
  PIMD_ERRNO_COMMAND_LIMIT_REACHED,

  // Command is not done
  PIMD_ERRNO_NOT_DONE,

  // Connection has max unretired recvs in flight
  PIMD_ERRNO_MAX_UNRETIRED_CMDS,

  // This is return on an open()
  // if create of a pds is requested and that pds exists
  PIMD_ERRNO_PDS_ALREADY_EXISTS,

  // open() could not find a pds by a specified name
  PIMD_ERRNO_PDS_DOES_NOT_EXIST,

  // Inconsistent permissions on pds open()
  PIMD_ERRNO_PERMISSION_DENIED,
```

```
// it_post_send() did not return IT_SUCCESS
PIMD_ERRNO_IT_POST_SEND_FAILED,

// it_post_recv() did not return IT_SUCCESS
PIMD_ERRNO_IT_POST_RECV_FAILED,

// it_post_rdma_write() did not return IT_SUCCESS
PIMD_ERRNO_IT_POST_RDMA_WRITE_FAILED,

// Out of memory for a command
PIMD_ERRNO_OUT_OF_MEMORY,

// Key does not exist in the database
PIMD_ERRNO_KEY_NOT_FOUND,

// Key too large
PIMD_ERRNO_KEY_TOO_LARGE,

// Value too large
PIMD_ERRNO_VALUE_TOO_LARGE,

// Record is locked
PIMD_ERRNO_RECORD_IS_LOCKED,

// Data intended for control message comm is too large
PIMD_ERRNO_CONTROL_MSG_LIMIT_EXCEEDED,

// Feature not yet implemented
PIMD_ERRNO_NOT_IMPLEMENTED,

// End of records
PIMD_ERRNO_END_OF_RECORDS,

// Streaming cursor ran out of buffers
PIMD_ERRNO_NO_BUFFER_AVAILABLE,

// Server data cursor reached the end
PIMD_ERRNO_CURSOR_DONE,

// Input buffer is too large
PIMD_ERRNO_BULK_INSERT_LIMIT_EXCEEDED,

// The limit of outstanding commands is set by PIMD_MAX_COMMANDS_PER_EP
PIMD_ERRNO_PENDING_COMMAND_LIMIT_REACHED,

// Internal use: communication buffer checksum mismatch
PIMD_ERRNO_CHECKSUM_MISMATCH,

// Internal use: streaming cursor iterator end of buffer reached
PIMD_ERRNO_END_OF_BUFFER,

// Cursor rank is out of range
PIMD_ERRNO_INVALID_CURSOR_RANK
} pimd_status_t;
```

Bibliography

[1] OpenFabrics Alliance. `http://www.openfabrics.org/`.

[2] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vocking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*, 35(6):1350–1385, 2006.

[3] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[4] Blake G. Fitch, Aleksandr Rayshubskiy, Michael C. Pitman, T. J. Christopher Ward, and Robert S. Germain. Using the active storage fabrics model to address petascale storage challenges. In *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 47–54, New York, NY, USA, 2009. ACM.

[5] R. F. Freitas and W. W. Wilcke. Storage-class memory: the next storage system technology. *IBM J. Res. Dev.*, 52(4):439–447, 2008.

[6] Bernard Metzler. `http://gitorious.org/softiwarp`. IBM Zurich Research Laboratory.

[7] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.