# Forwarding IO with Portals in PVFS

Project Report

Pete Wyckoff
pw@osc.edu

August 31, 2007

## 1 Summary

DoE funded a small project to port PVFS to Portals, the messaging layer used on Cray XT3 and XT4 platforms. Design, coding and testing on the TCP implementation of Portals has been completed successfully. PVFS compiles on the Cray platform, but we are struggling to find willing parties to let us test on their Cray systems. There is potential to extend the scope of the work plan to perform testing and performance characterization on large scale systems. This future work would benefit other I/O architectures on large scale systems as well.

## 2 Background

The PVFS file system is a production-ready parallel file system for use on high-end computing platforms. Version 2 of PVFS is routinely deployed in production on both Linux cluster systems and on the IBM BG/L platform. Its design is modular, allowing support for new networks and storage devices as they become available.

Leadership class computing facilities are deploying cutting-edge systems at unprecedented scale. As such, few parallel file system options are available for these systems. On the IBM Blue Gene systems, for example, only Lustre, GPFS, and PVFS are currently viable parallel file system options. On the Cray XT3 platform, Lustre is currently the only parallel file system option.

In order to mitigate risk, and to enable evaluation of parallel I/O options on the Cray XT3 system, making a second parallel file system option available on the XT3 platform is very appealing. Because of qualities of the PVFS design, such as its modularity, porting PVFS to new systems is relatively simple. This makes PVFS a compelling candidate as a second parallel file system option for the XT3. Further, because PVFS is community developed, open source, and freely available, porting PVFS to the XT3 would provide all sites deploying XT3 systems with a no-cost risk mitigation strategy for the parallel file system.

We proposed to port the PVFS parallel file system to the Cray XT3 system.

## 3 Design and Implementation

The bulk of the work was in designing, coding and testing a new network interface for PVFS that would use portals. Discussions with Ron Brightwell and Lee Ward of Sandia National Labs were helpful in deciding how to use the Portals API effectively in this situation. The more interesting design decisions are described in the following sections.

### 3.1 Connection management

Portals is (for the most part) connectionless. For processes to find each other requires external support. Three bits of information are required to communicate: nid, pid and portal index. The BMI address will provide a hostname and the pid. We assume some external mechanism to convert hostname to nid; for the utcp portals device this is the IPv4 address. The pid is explicitly provided in the BMI address. The portal index for the server is “well known” and hard-coded in the program.

There is no connection structure in the code. Duplicate requests are never generated. We rely on the reliable nature of Portals to avoid this issue. Messages may be dropped in Portals, but the application always knows how each message completed.

### 3.2 Data motion

Portals has the same mismatches with BMI that InfiniBand (IB) has. The crux is that BMI permits sends to occur without pre-matching receives, but Portals would drop these messages. The sender always sends the entire message, and if the receive happened to be preposted, everything is done then. Portals sends an ack back to the sender under the hood, indicating completion. If not preposted, the data goes into one of a few big buffers, but only the first so much of the data is retained. If the data fits completely, the sender is done. Later when the receive is posted, if the full message happened to have been received, it is copied out of the unexpected area and all is done. If the message was truncated, the sender knows this and arranges things so that the receiver can do a Get. Events happen and all completes. The Portals atomic `MDUpdate` command is used to handle the usual unexpected race condition.

Note that the other usage of the word “unexpected” message in the BMI layer is for the method calls `post_sendunexpected` and `testunexpected`. These are essentially new requests from clients, and never really have an explicitly pre-posted receive, but need to go somewhere that the server can find them. We manage a queue of preposted buffers for all possible senders, not a separate queue for each sender as in the IB case. (IB was written before SRQ implementations were available.) There is no explicit flow control, so if the clients manage to overwhelm this buffer, we rely on the upper layers of PVFS to cancel the request and try again. Adding resends in BMI would lead to duplicate messages and require adding per-client state. Limiting the number of outstanding requests per client, and sizing the buffer on the server to expect that many may guarantee that this never happens.

Put buffers and get destination buffers do not have to be explicitly registered. Building the Portals MD structures does that under the hood, if necessary.

### 3.3 State paths

Below are descriptions of how the states progress for the sender and receiver for the various possible message types.

```
BMI_post_send
    start
        do the put operation
    SQ_WAITING_ACK
        (ack arrives)
        mark completed, transition
    SQ_WAITING_ACK
        (timeout occurs)
        redo the put operation, stay in this state
    SQ_WAITING_USER_TEST
```

```
        (user tests)
        release sendq
BMI_post_recv
    start
        build md/me
    RQ_WAITING_INCOMING
        (event happens)
        mark completed, transition
    RQ_WAITING_USER_TEST
        (user tests)
        release recvq
```

The function `BMI_post_sendunexpected` is identical to its expected counterpart on the sender, but the put operation uses different match bits and specifies a receiver-managed offset.

## 3.4 Event processing

All Portals message descriptors produce events to a single event vector (EV). We distinguish among them by the user pointer field provided by Portals. This pointer is upcast to the appropriate outstanding message structure to advance the state machine.

## 3.5 Match lists

The match list is an important element in any Portals application. It is the sequence of match entries and memory descriptors that will be encountered by a new incoming message. This diagram shows the match list used in PVFS.

```
preposted receives
    match bmi_tag → preposted buf
    match bmi_tag → preposted buf
    ...

outgoing sends
    match 2 << 32 | bmi_tag → preposted buf, respond to get
    ...

mark
    match none

nonpreopsted receive buffers
    match 0 << 32 | any → nonprepost buffer1, max size
    match 0 << 32 | any → nonprepost buffer2, max size

unexpected message buffers
    match 1 << 32 | any → unexpected buffer1, max size
    match 1 << 32 | any → unexpected buffer2, max size

zero
```

match $0 \ll 32 \mid \text{any} \rightarrow$ no buffer, trunc, max size 0

Preposted receives must come first and be in order so that they match for expected incoming messages. The order of nonprepost and unexpected buffers does not matter, so we let them mix up among themselves. The mark entry is used to be able to find the point between the prepost and other entries, otherwise we would need lots of code to track that by hand.

The nonprepost and unexpected buffers are managed as "circular" lists, where one is filled up until it is unlinked, then it is reposted after the other that has now started to fill up.

Nonpreopst messages are kept in the buffers until the app posts a receive that matches. If they fill up, later messages fall off the bottom. Working apps will pre-post their receives before the sender tries to send to them.

Unexpected messages are a protocol feature of BMI. A special high-bit indicates this. They are limited in size by the protocol (8 kB here), and are always new requests from a client to a server. As they arrive, they are immediately copied into new buffers that are handed back to the server through `BMI_testunexpected`. So we do not need reference counting and the circular buffer management is straightforward: just repost immediately when the unlink event happens.

Finally, for nonprepost messages that are too big, it lands into the zero md at the end that just generates an event on the sender and receiver. The receiver does a get to read the data from the sender later when the app finally posts the receive.

### 3.6 Possible improvements

Performance might be better to have separate completion queues for sends and receives. Currently accesses to the single queue must be locked by the send and receive threads that wish to process them.

Outstanding message structures are linked together in a standard circular list. This leads to $O(N)$ lookups for some operations, including incoming RTS (if used later) and receive matching. A tree-like data structure or hash table may be better.

All messages are sent in "eager" mode, where the entire message is transmitted before ensuring that the receiver is ready for it. This is true even for very long messages. The advantage of doing so is that it optimizes for pre-posted behavior. While this is provably the case with many PVFS operations, there are situations where the message will be dropped by the receiver, and then re-transmitted as the response to a get operation from the receiver. Another long message protocol option is to send a short "ready to send" message, then have the receiver issue the get when it is ready, either immediately or once the receive has been posted by the application. This may avoid overloading the network with wasted traffic, but requires an extra round-trip on the network for each long message, even if it was already pre-posted.

## 4 Summary

This work is fully integrated into the current PVFS source distribution. It includes the 2000 lines of C code that implement Portals networking, an extensive README file describing the implementation, and hooks into the various file system configuration and status tools to understand the new Portals interface.

Extensive benchmark and application testing on the TCP version of Portals was used to ensure that the code functions correctly. However, performance is no better than the underlying TCP communication layer, and the Portals implementation adds significant overhead due to some design constraints with using TCP. Performance on hardware that supports Portals natively should be much better.

We made modifications to the PVFS build system and fixed various porting issues to be able to compile on the Cray XT3 at ORNL. The code has not been tested on the ORNL machine. The main impediment is that the system administrators there are hesitant to let us run Portals code on the service nodes of the system. This is necessary as we need a more extensive operating system to support the PVFS servers, which require threading, sockets, and the Berkeley database, as well as a local file system into which to put data.

Ruth Klundt and Lee Ward of Sandia are in the process of creating an account on their 80-processor Cray XT3 for PVFS testing. We plan to use that machine to work out any bugs in the PVFS implementation, and to ensure that it does not cause any adverse affects to other jobs or to the service nodes. With that information, hopefully the ORNL administrators will be convinced that the code is safe to run at larger scale on the machine there. If all goes as plans, this will allow collection of performance numbers and the start of a future possible optimization project.

## 5 Project Continuation

While we have completed the milestones in the original project plan, we were unable to perform any performance optimizations on real Portals hardware, namely the Cray XT3 or XT4. Beyond the end of the project, we will ensure that the code does function on those machines, but will not be able to expend the effort to make it perform well.

With increased support, it would be possible to accomplish a number of further goals. First, performance optimization of PVFS on the Cray systems as mentioned previously. This includes the issues of the long eager protocol, receive matching with large numbers of peers, and event queue locking issues, as discussed in the Future Work section above. Work in this area would benefit all large scale machines, not just the Cray, and would naturally improve performance of PVFS on BG/L machines.

Second, integration of PVFS into the POSIX messaging libraries of the Cray would allow use of PVFS by applications that perform I/O with the C library rather than with MPI-I/O. This would expand the set of applications and users that can take advantage of high-speed I/O on the Cray, and would also apply to other and future large-scale machines.