



Fermi National Accelerator Laboratory

FERMILAB-Conf-96/002

**POPM: A Distributed Query System for High Performance Analysis
of Very Large Persistent Object Stores**

Mark S. Fischler, Michael C. Isely, Ariel M. Nigri and Frank J. Rinaldo

*Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510*

January 1996

Submitted to the *Hawaii International Conference on System Sciences*, January 1996

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

POPM: A Distributed Query System For High Performance Analysis Of Very Large Persistent Object Stores¹

Mark S. Fischler², Michael C. Isely³, Ariel M. Nigri⁴ and Frank J. Rinaldo⁵

Fermi National Accelerator Laboratory

P.O. Box 500

Batavia, IL 60510 USA

Submitted 5/29/95 to the Hawaii International Conference on System Sciences, January 1996

-
1. This work is supported by the U.S. Department of Energy under Contract No. DE-AC02-76CH03000.
 2. mf@fnal.gov; (708) 840-4339 fax: (708) 840-8208
 3. isely@fnal.gov (708) 840-2784
 4. nigri@fnal.gov (708) 840-3819
 5. rinaldo@fnal.gov (708) 840-8449

Abstract

Analysis of large physics data sets is a major computing task at Fermilab. One step in such an analysis involves culling "interesting" events via the use of complex query criteria. What makes this unusual is the scale required: 100's of gigabytes of event data must be scanned at 10's of megabytes per second for the typical queries that are applied, and data must be extracted from 10's of terabytes based on the result of the query.

The Physics Object Persistency Manager (POPM) system is a solution tailored to this scale of problem. A running POPM environment can support multiple queries in progress, each scanning at rates exceeding 10 megabytes per second, all of which are sharing access to a very large persistent address space distributed across multiple disks on multiple hosts. Specifically, POPM employs the following techniques to permit this scale of performance & access:

Persistent objects: Experimental data to be scanned is "populated" as a data structure into the persistent address space supported by POPM. C++ classes with a few key overloaded operators provide nearly transparent semantics for access to the persistent storage.

Distributed & parallel I/O: The persistent address space is automatically distributed across disks of multiple "I/O nodes" within the POPM system. A striping unit concept is implemented in POPM, permitting fast parallel I/O across the storage nodes, even for small single queries.

Efficient Shared access: POPM implements an efficient mechanism for arbitration & multiplexing of I/O access among multiple queries on the same or separate compute nodes.

The internal POPM engine implementation is optimized from the ground up to employ principles of overlapped, multiple I/O, along with an aggressive read-ahead algorithm. This results in higher utilization of available bandwidth, lower effective latency, and better overall performance than a comparatively naive query engine design.

The POPM system is currently targeted for use in the multi-node IBM SP architecture. By using the high speed switch, overlapped I/O, and multiple compute nodes, the system will be able to sustain the desired high throughput required to access the distributed persistent address space.

1.0 Introduction

There are many applications in computing which require "data mining". The culling of "interesting" physics events from experimental data here at Fermilab is an example of such a problem. Another example includes the scanning of large customer purchase records in order to learn of buying trends. Problems of this nature abound.

The trouble with the data mining task is that the numbers involved are extremely large: In our case, the source data is terabytes in size, and searches must complete quickly (tens of megabytes per second) in order to be useful. Efficiently focusing enough compute resources on this type of task is a non-trivial problem.

To attack the data mining task requires defining and addressing two problems: (1) Exactly what sort of software model can present an efficient representation of this data? (2) How does one design a system that can implement such a model and at the same time work quickly and efficiently in the face of such large data sets?

The nature of the task, both in size and organization, simply defies any attempt at mapping into a traditional row & column database paradigm. Thus we apply persistent object-oriented database concepts to our problem. The actual model we use follows the PTool concept API [1][2], developed at UIC as part of the PASS project [3].

This leads to the second question: How do we make this function on such a large scale? One can't simply attach terabytes of disk to the typical workstation, run PTool, and expect reasonable query times, even for a single query operation. Data sets this large require distribution of the data over many processors, and the application of some kind of parallelism when performing the scans [4]. Our solution is the Physics Object Persistency Manager (POPM) system, deployed onto the IBM SP distributed architecture as the computing platform. The main focus of this paper is POPM.

This paper covers each of the above points. First is a description of our particular data mining problem, along with an overview of the PTool persistent object API and how it is used to represent our data efficiently. Following that is a description of the problems and issues associated with running over a distributed architecture.

The paper then goes on to demonstrate the POPM system concepts. Specifically, it will be shown how POPM presents an efficient implementation of the PTool object model over the distributed architecture of the IBM SP (Scalable Power Parallel) hardware - both by extending the API in strategic places, and by exploiting natural opportunities for parallelizing I/O during typical queries.

Expected and measured performance of POPM is discussed - including single process scanning rates, and the designed scalability when multiple queries share access to the common global data.

The final section puts all the pieces together. It will show how this system will solve our data mining problems at Fermilab, and in fact points out that POPM can be applied to other more general tasks involving data mining problems.

2.0 Problem Description

2.1 Event oriented data & objectives

Data mining is the process of finding patterns and relations in large datasets. The term 'data mining' is used because the typical procedure involves processing very large amounts of data (i.e. ore) to extract a few 'nuggets' of information (i.e. gold).

A typical data mining application would be in the area of consumer goods analysis. Here dataset records correspond to a shopping 'basket'. The information being extracted would be buying patterns - relationships between particular goods or classes of goods (e.g. fertilizers and fuel oils). This processing is very different from traditional database applications where some fields in a record are declared as primary (and/or secondary) keys so that an index can be built for quick retrieval of a particular record (or a small subset of records). Data mining applications require quick access to an arbitrary set of records.

At Fermilab the data record ('shopping basket') contains details describing a particle interaction event - information about one collision between a proton and a target or an anti-proton. The nuggets of information that we are searching for are events of particular interest, e.g. Top Quark candidates, defined by complicated relations among the event data. The datasets at Fermi range from 10 Terabytes to hundreds of Terabytes. The ability to 'sift' through this much data requires an innovative solution involving high-performance computing, parallel I/O, and a new software paradigm. The strategy chosen is to divide the data into collections of persistent (magnetic storage-resident) objects, and scan only those sets of objects necessary to the query at hand. This paper focuses on the generic light-weight persistent object manager software that has been developed to support data mining applications.

Within each High Energy Physics event (each record), the data can be organized into attributes of several "physics objects" such as particles. An event can have an arbitrary number of each type of object, so treating each event as a row in one large table, as would occur in a relational database, is impractical. Alternatively, entire events may be stored compactly, but a scheme requiring retrieval of all of the data for each event processed is unnecessary and undesirable.

Instead, the data can be organized as collections of objects of each type. Then efficient access patterns can be achieved by scanning only the types of objects involved in the query. The decisions about what constitute each collection of objects are flexible; here expert users can inject common-sense knowledge of the nature of the problem. But with this data grouping, usual database techniques become impractical for queries involving multiple objects. A typical physics selection might contain such criteria as

```
electron#1.E + electron#2.E > 25
```

This implies a double loop, but the second loop need not range over the millions of electron objects in the data: Since each event is independent, this loop is over the several electrons in each event.

We call this property event-organization. In such a data set, query software must take advantage of this organization to run efficiently - in the above example, any method requiring creation of all pairs of electrons in the entire dataset, or of an intermediate file of size comparable to the electron data, cannot be practical. Physics data is not a lone example: Consider market-basket data. Each purchase item is described as an object; the entire basket is an event. A sample query, to create data relevant to comparison-shopping habits, might look like:

```
(cereal#1.coupon != 0) && (cereal#2.coupon == 0)  
&& (cereal#1.brand != cereal#2.brand)
```

This query requires examining every pair of cereal purchases within a single event, rather than every pair that can be formed out of all purchases.

Another instance of event-organized data is credit transaction histories. Here each person is an "event", in the sense that queries select a set of people with specified combinations of transaction properties. The objects are individual transactions - unlike the physics and market-basket examples, in this case data added to a database would tend to be additional objects added to existing "events". But the key property - that queries relating multiple transactions loop only within the person's history, and not the entire database - still holds.

2.2 PTool style software modeling strategy

To avoid reading in all of the data for a query concerned about only a few types of physics objects, all objects of one type should be stored together. But the code determining whether an event is to be selected will occasionally require not only information about the first object mentioned in the query, but about physics objects of other types within an event. Logically, the entire event can itself be viewed as an object, although the various physics objects associated with it are stored in disparate places. To support queries relating objects of different types

within an event, each object contains a pointer to the event-object; the event-object has pointers to the physics objects of various types that make up that event.

Since this data resides not in memory but on disk, these "pointers" are not the familiar memory-address pointers. They must remain valid across the lifetimes of processes - an executing query process uses pointers to objects in the database which it neither has created nor explicitly read into memory.

These special pointers are based on the concept of a "persistent object". This is a C++ structure with the property that the principal representation of that data resides in permanent storage (disk or tape) rather than in memory. Of course, when the data is used, it will be brought into memory; but this is logically just as transparent as main memory being brought into cache when the CPU needs it. The syntax for working with persistent objects is defined by the Ptool API, which employs the concept of persistent pointers (Pptr).

The persistent pointer is implemented as an Abstract Data Type with overloaded operators for dereferencing, memory allocation, comparison and assignment (->, new, ==, !=, and =). In both POPM and the original PTool implementation, persistent pointers to specific classes are defined by deriving a class from the base implementation, and thus have the advantage of providing type checking.

A Pptr can be dereferenced to yield a member of a persistent object, just as an ordinary pointer can be dereferenced to yield a member of an ordinary object. The POPM class library supporting this API implements this by overloading the "dereference" (->) operator for persistent pointers. This will provide an l-value for the specified member of the object. When a Pptr is dereferenced, POPM checks whether the corresponding persistent object is in memory, and if not, causes a chain of events involving the local processor, remote processors, and disks and/or tape libraries, to read in the data and return the required l-value.

When the overloaded "new" operator is invoked to get a persistent pointer, a persistent instance of that class is created, and the programmer receives a Pptr pointing to it. Data stored using that Pptr will persist on disk past the end of the process. This is how data is loaded into the persistent object database. The "new" operator has one argument - the name of a "store" which defines an address space to group related objects. The Ptool object model requires persistent objects to be grouped into stores, so that the storage can be organized to optimize sequential access to a store.

In order to keep the implementation of this persistent storage space simple and efficient, boundaries are established so that objects will fit in a reasonable piece of the process address space. The semantics

implementing this is the concept of "segments." An object, to be instantiated as persistent, must fit in a single segment, although a single segment may contain many objects. This approach simplifies the I/O subsystem, which can work with segments rather than objects. The segment size is determined by a compromise among I/O performance considerations, the sizes of large objects in our applications, and the desirable flexibility of having many segments in core at once. It is currently set to 64 kBytes.

2.3 Hardware Architecture

One of our major design goals was to get the most performance for the least amount of dollars. This goal is pursued by using commodity devices (disk, SCSI bus, tape storage, processors, etc.) in a scalable parallel system. Multiple disks are placed on multiple SCSI buses on multiple processors to maximize throughput at a minimal cost. The idea is to have scalable massively parallel I/O on processors dedicated/optimized for this task. These I/O processors then pass the data on to multiple parallel compute (CPU bound) processors for the actual data evaluation and extraction. Besides having a particular query execute across many I/O nodes, we also need to allow multiple queries to execute in parallel on the system. This is to maximize utilization of the overall system and take advantage of various levels of optimization and caching.

Our current prototype [5] is based on IBM's SP-2 Power Parallel system. The SP-2 architecture is a distributed memory, message passing parallel processor system. Message passing was felt to be better than shared memory for scalability to large number of processors [4]. Each node in the system is based on the RS/6000 processor. These nodes can be configured to be compute nodes and/or I/O nodes. All nodes are interconnected by a proprietary high-speed switch which allows all processors to send messages simultaneously. The prototype's peak bisectional bandwidth is rated at 272 MBytes/sec.

One of the limitations of the SP-2 system is that only one process (per node) is permitted access to the high-speed switch. At first glance this would mean that only one "job" (i.e. user based query) could run on the entire system at a time. Each user would lock the entire system's access to the high-speed switch until its query was completed. Our software must address this problem by allowing many users to execute simultaneous queries distributed across the entire system via the high-speed switch thereby maximizing utilization and throughput. The software also needs to distribute the user's data across the I/O nodes, SCSI buses, and disks in an efficient manner to maximize parallel access.

Our prototype is currently a 24 node system, configured as 8 I/O nodes and 16 compute nodes. Each I/O node is capable of supporting multiple SCSI buses. The application data is distributed across I/O nodes, SCSI buses, and disks to maximize throughput via parallel I/O. The Performance

section will discuss the balance necessary for high efficiency of this system. Parameters include: number of I/O nodes vs. compute nodes, SCSI buses per I/O node, SCSI disks per SCSI bus, number of simultaneous queries, etc.

3.0 POPM Implementation

The PTool object framework provides a model which fits our problem, and storage distribution across many CPUs makes possible the required high capacity and fast throughput. The software which implements the PTool object model on the distributed architecture is POPM. The following sections explain the POPM implementation concepts:

1. How PTool data is related to fixed segments, and how POPM stores those segments.
2. The basic POPM concepts, relating to a "trivial" single CPU system.
3. The full multiple CPU POPM implementation. The implementation on the IBM SP-2 distributed system is described.

With the overall concepts presented, specific parts of POPM are highlighted, which show where the POPM implementation takes advantage of the high performance capabilities of the distributed IBM architecture. Also described are enhancements POPM presents to the persistent object API as originally defined in PTool.

3.1 Basic Implementation Concepts

The smallest unit of storage in the PTool model is the segment; this has a fixed size set to 64 kBytes. When the overloaded "new" operator is used to create persistent objects within a store, segments are created as needed for the underlying persistent storage. PTool stores are in fact composed of sets of these segments.

Mapping of a PTool object into its assigned segment is a fairly easy task; the POPM class library, against which POPM programs are compiled, handles that function transparently and efficiently. The difficult problem is the implementation of fast, efficient segment transfer into the memory of the process doing the data mining - an operation that must scale well with the number of queries in progress. The POPM implementation is specifically designed to address those issues.

The original UIC PTool class library performed segment I/O via mapping: As segments were needed, each was directly mapped from its file location into one of several possible "slots" in a fixed size array of segment-sized slots in the process's transient address space. If all slots were mapped, the oldest one was picked and remapped to the required segment. This array logically behaved as a "cache" of the last few accessed segments, and makes possible semi-transparent access to all persistent data in a PTool program.

The POPM implementation does things differently. The cache concept is still employed, but because the segment may be located on a CPU other than the one running the query, mapping can't be used. In addition, allocating a private slot cache inside the program's address space can be wasteful if more than one query is running on a given node. POPM addresses both of these issues through two new concepts [6]: (1) A single CPU-global shared memory slot cache is mapped by all query programs. (2) Segment transfer to/from the shared slot cache is handled by separate "disk slave" daemon processes.

In addition to several obvious advantages (dynamic load balancing, low copy overhead), designing POPM around this shared slot cache concept enables easy implementation of a very useful goal: disassociation of the segment transfer activity from the actual query processing. Segments are transferred not by the query process, but by one of several disk slave processes running as part of POPM. When a query process requires a particular segment, all it needs to do is allocate a slot, and set up a request for the transfer in that slot's header. A disk slave then picks up the request and performs the transfer. This separation of the act of segment transfer from the query operation creates opportunities for significantly increased performance, including overlapped I/O, and transparent distribution of segment data (discussed later).

A POPM system can run multiple disk slaves - each is a server just looking for work to do. When a slot has a pending segment I/O request, any disk slave may service it. Multiple pending requests are naturally divided among the disk slaves, resulting in implicit parallelization of I/O. Thus a single query process may achieve faster parallel I/O simply by setting up multiple slots for transfer.

In addition to disk slave daemons, and the query processes themselves, there is one other local daemon needed for POPM: the shared memory manager. This process performs global "watchdog" functions for the shared slot cache: It attempts to balance slot usage among query processes, and it performs any wreckage cleanup / integrity verification if any other processes should happen to fail or crash. The shared memory manager daemon does not participate in actual segment transfers. Logically, the shared memory manager can be thought of as an active part of the shared memory slot cache; it handles any house-keeping activities needed on behalf of the system.

Here is a diagram illustrating all of the concepts described thus far, which is enough to run POPM on a single node:

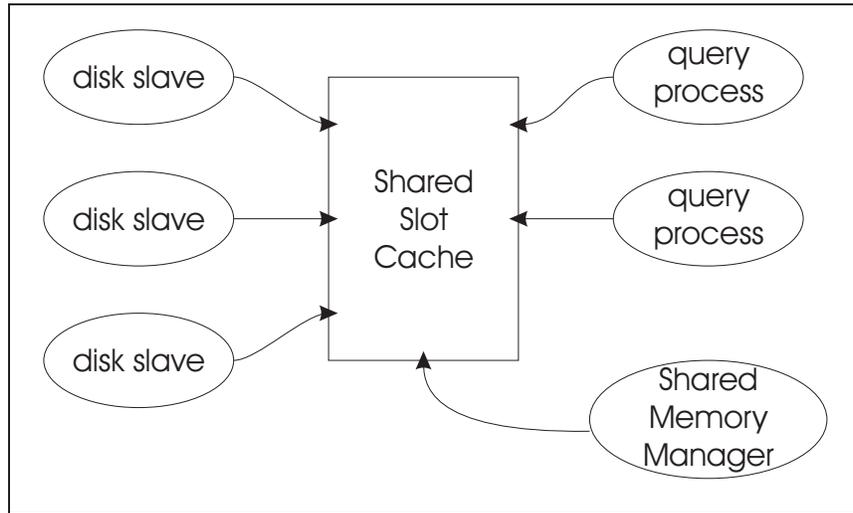


Figure 1. Single CPU POPM system

The above description illustrates a POPM system configured for a single node. For only one node, one does not really need such a complex scheme for performing I/O. This extends easily, however, to form the foundation for a multiple node configuration.

To create a multi-node POPM system, the single node system is replicated across all nodes and a new process type is introduced for inter-node segment transfers: The I/O server process.

Every host in a multi-node POPM system has the basic components of the single node system: a shared slot cache and a shared memory manager. Nodes which contain persistent storage (I/O nodes) also run disk slaves. Query processes run on designated compute nodes. And each node runs an I/O server process.

All of the I/O server processes in the system function together as a unit, coordinating via the MPL user-space interface to the SP-2 high speed switch. They cooperate to forward slot requests to the proper nodes containing the requested segment(s), and return the results. I/O servers therefore logically behave as surrogate disk slaves on nodes with query processes, and as surrogate query processes on nodes with disk slaves. I/O server processes run continuously; the I/O servers effectively make remote disk access appear local to the node(s) which require it.

Following is a diagram showing an example 4 node POPM system:

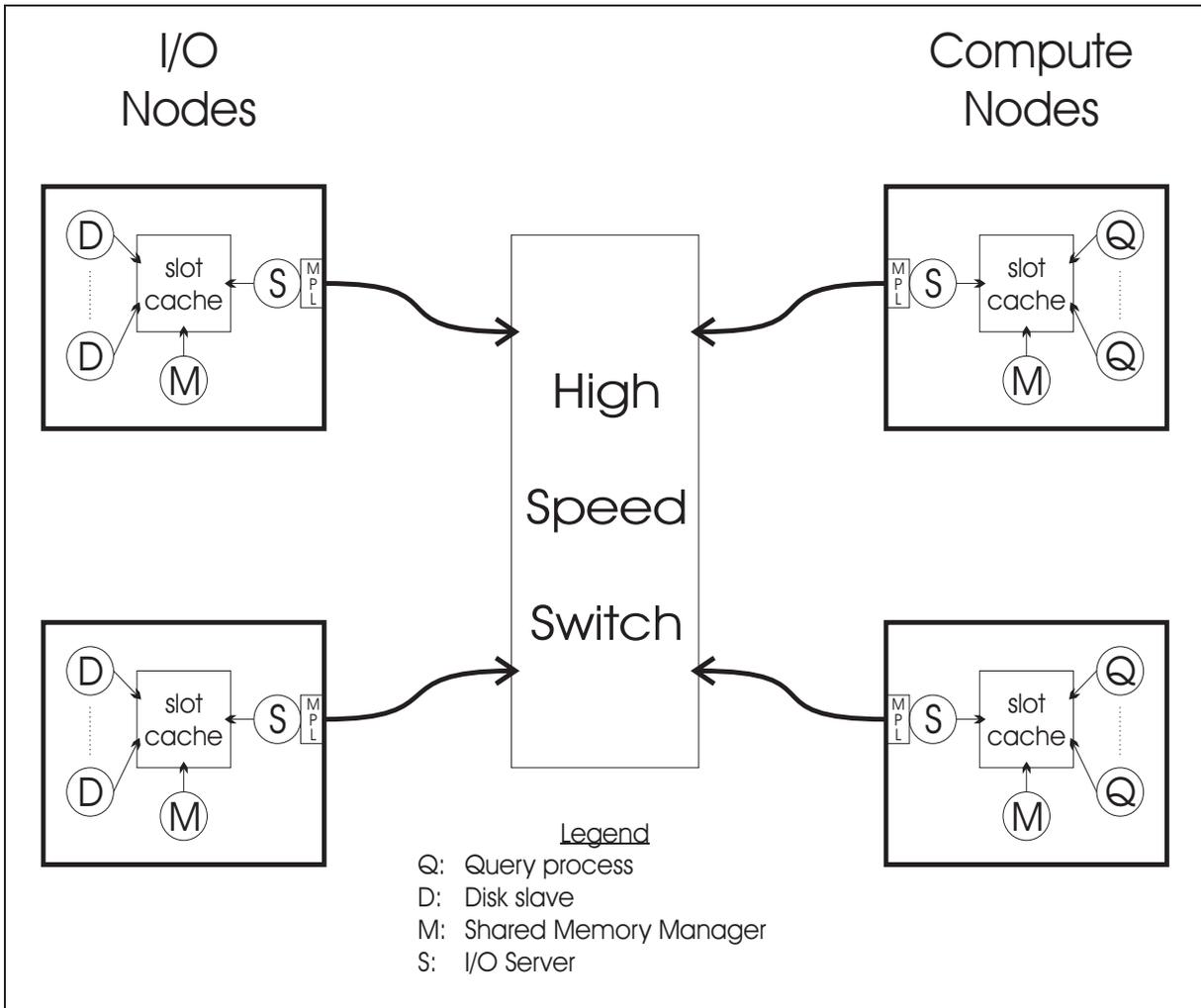


Figure 2. Multi-node POPM configuration

The I/O server implementation cleanly compartmentalizes all access to the switch interconnect (the "MPL" boxes in the diagram are IBM's *Message Passing Library*), through a carefully designed class interface. Porting POPM to a new distributed architecture is simply a matter of changing this class implementation.

Note that this I/O server based approach means that only one access point into the switch is needed per node, no matter how much local POPM activity is taking place. This makes possible the use of distributed architectures with non-shareable interconnect access, like the SP-2 user space switch interface, for multiple query processes.

3.2 POPM Performance Issues

The above describes the overall concepts relating to structure of POPM with a few hints about performance. Now we delve into some of the details which enable efficient operation. POPM employs overlapped I/O, striping

techniques, and intelligent read-ahead. These three concepts, when combined, have a synergistic effect providing the fast single process query throughput require by our problem.

A pending segment I/O request in POPM is stored as control information associated with the slot cache element for the request; this means that there can be as many pending requests as there are slots in the system. In addition, the I/O server processes are designed in a stateless fashion; the forwarding of I/O requests and retrieval of the results are separate operations with no intervening blockages. This means that I/O servers can handle effectively arbitrary numbers of I/O requests at one time. Combined with the fact that all disk slaves operate independently, this lets the entire segment transfer path behaves like a pipeline. All stages work independently from each other, from the query process through the shared slot cache and the interconnect, all the way to the disks. Segment transfer requests can be processed in parallel, and generally won't have any limiting serialization dependencies.

The striping aspects of POPM are affected by the way segments are stored: The logical organization of a store is a sequential collection of segments, but the physical organization is different: A store is contained as a collection of files, called folios, on the persistent storage medium. An individual folio, in turn, is composed of a fraction of the segments for the store - folios within a store are of equal size. When POPM needs to access a specific segment of a particular store, it performs two computations: First it finds the folio in which the segment is mapped, then it locates the storage device containing that folio and accesses the proper offset within that file to reach the segment.

POPM striping is a matter of properly defining the mapping pattern of segments to folios (segment striping), and of folios to storage devices (folio striping), such that access performance is optimized. With two storage devices and four folios, a naive mapping might be:

folio number:	folio1	folio2	folio3	folio4
segment numbers:	seg1	seg4	seg7	seg10
	seg2	seg5	seg8	seg11
	seg3	seg6	seg8	seg12

Figure 3. Naive segment assignment

storage device:	disk1	disk2
folio:	folio1	folio3
	folio2	folio4

Figure 4. Naive folio assignment

If a block of segments are requested at once, throughput will be enhanced if the request can be parallelized across the storage devices. The above mapping won't produce that behavior unless the block of segments requested happens to span the boundary between the 2nd and 3rd folios. Striping solves this problem:

folio number:	folio1	folio2	folio3	folio4
segment numbers:	seg1	seg2	seg3	seg4
	seg5	seg6	seg7	seg8
	seg9	seg10	seg11	seg12

Figure 5. Striped segment assignment

storage device:	disk1	disk2
folio:	folio1	folio2
	folio3	folio4

Figure 6. Striped Folio assignment

Blocks of sequential requests - as small as 2 segments - will be implicitly spread among the storage devices. Since multiple storage devices will be accessed, POPM's pipelining and overlapped I/O architecture will come into play and parallelize the entire operation. But how does one cause a single query to request multiple segments at once? That is addressed by the POPM read-ahead mechanism.

A typical query process working through event-oriented data is going to be sequentially scanning through at least one set of objects. For any given set, the data may be dense, resulting in sequential scanning. This sequential behavior is something that can be automatically detected and exploited by the query process. The algorithm is, in concept, very simple: consider each open store a possible read-ahead "stream" and maintain additional internal state for it. This state would include the last segment accessed. When an access requires a segment that is not the same as the last segment accessed, check if the required segment is the next one in sequence. If this "tends" to be true, then there is a read-ahead opportunity. Perform the read-ahead by allocating a few more slots and pre-requesting the "next few" segments. The proper definition of "next few", or the read-ahead depth, can be automatically tuned by keeping track of the current depth and noticing whether or not the "next" segment is present by the time it is needed. If it isn't, POPM will increase the depth. If it is, the next few segments are also checked for readiness; if "too many" are already present (where "too many" is derived from the variance in the average latency to fetch a segment), POPM will decrease the depth in order to conserve slot resources.

An absolute maximum read-ahead depth, based on the number of read-ahead streams that may be active on the node, is tracked in order to prevent one I/O-bound stream from using all slots and starving out other read-ahead streams. The shared memory manager participates in the read-ahead depth computation to ensure sensible allocation among the streams.

This strategy results in a self-tuning behavior in which effectively only enough I/O bandwidth to "keep up" is actually dedicated to the read-ahead stream. Result: efficient bandwidth allocation among competing read-ahead streams.

With read-ahead in place, POPM is able to block-request multiple segments at one time. Since POPM is able to process these requests in parallel, the overall performance can potentially be very high, even for a single query process.

3.3 POPM functionality enhancements

In addition to transparent performance enhancements, POPM adds some important functionality enhancements to the original PTool API definition. Namespaces make the persistency model more workable in a large scale production environment. And locked persistent pointers, or "physical pointers", provide an important refinement to the PTool API.

In the original Ptool API, which is inherited in POPM, there is the concept of a single global 64 bit persistent address space. This is implemented by maintaining a global database of all stores within the address space. This would be fine if the entire system were dedicated to scanning only one persistent address space. But in general, there will be several production datasets (in our case data from different physics experiments), and there will also be test datasets, each of which should be contained in its own persistent address space and have its own name space (with its own set of administrative protections).

POPM provides for multiple address and store name spaces, each of which is independent, but all of which may be concurrently accessed in a single system. POPM coordinates these multiple address spaces by supporting multiple databases for stores - one database per address space.

A individual POPM query is still confined to one address space: It must be "declared" at start-up, after which all accesses are confined to that space. When a query process declares its address space, the corresponding database is selected by POPM for all further references.

Access to PTool API persistent data is normally accomplished through the use of the overloaded dereference operator ("->") for the Pptr data type. Every call results in a search of the slot cache (POPM uses a hash look-up) in order to find the "real" address through which object data can be currently reached. On the surface, this seems reasonable, but there are two important concerns which must be addressed: (1) Every

persistent access, even at the finest granularity, will result in an expensive cache search. (2) There is no defined lifetime of validity for the computed real pointer returned by this search; another dereference may cause the slot to be replaced with other data.

While the repeated searches have only a performance impact, the pointer validity issue potentially affects correctness. Nothing prevents a compiler from applying the overloaded "->" operator to obtain a transient physical address, then loading data from that address sometime later (within the same code block). Indeed, if the dereferenced item is a member function that dereferences persistent data, this behavior is inevitable. Since even a naive implementation will provide memory cache slots for several persistent objects dereferenced by several Pptrs, this flaw has no practical consequences; but contrived examples can go astray.

Both the correctness problem and the performance problem are addressed by providing a way of locking a pointed-to slot and remembering the real pointer. In that way, access to multiple fields of an object would only cost a single dereference, and there is no fear of the underlying slot being replaced during that interval. POPM addresses this with a new "locked" pointer type in the API, called a "physical pointer". The semantics are that the user instantiates and assigns a physical pointer, and later either reassigns it, allows it to go out of scope, or explicitly destructs it when the data pointed to can safely be expunged. So long as the physical pointer is pointing at a valid persistent object, the corresponding segment is guaranteed to be present in a slot, and that slot is guaranteed not to be replaced by other segments. This explicitly defines the validity lifetime of any computed real pointer values. Also, since the dereference results are saved in the physical pointer instance, redundant cache searches for repeated accesses to the same object are avoided.

In POPM, slots can be replaced asynchronously with respect to the query process (as part of the load balancing algorithm in the shared memory manager). Unfortunately, this asynchronous behavior exacerbates the pointer validity problem for basic Pptrs. To insure that no problems can occur even in this case, POPM transparently applies the locked Pptr mechanism to the last N Pptr dereference operations (where N matches the slot cache size of earlier PTool implementations).

To better illustrate the performance benefit of physical pointers: A code segment utilizing several members of a persistent object via a conventional Pptr must, to be safe, dereference its persistent pointer each time. Setting an ordinary pointer to the address used by a Pptr (assuming it were possible) would seriously risk loading incorrect data. Each overloaded "->" has been measured on an IBM SP-2 system to take about 800 nsec (assuming the data is present). For procedures such as data-conversion, requiring every attribute of each object, this is a

serious issue. By instead employing a physical pointer, the code can address the various members of the persistent object attribute using an ordinary pointer:

```
int some_func(PPTR<mytype> pA) // mytype is some object defined elsewhere
{
    int result;
    LockedPPTR <mytype> xA(pA);
    mytype *A = xA;           // Assignment of physical pointer to ordinary pointer
    result = some_func (A); // data pointed to safely passed
                           // some_fucntion() can use many members of A
                           // and can itself freely dereference other persistent pointers.
} // Now xA is destructed, so lock is freed.
```

This avoids all but one dereference operation.

Having introduced a locking mechanism, the API must define the behavior when deadlock is threatened due to an excessive number of active locks. (For the same reason that unlocked Pptrs rarely have problems, such situations will tend not to happen unless the user errs by not releasing unneeded locks.) While recovery mechanisms are possible, programs routinely using such recovery would have horrendous efficiency. Worse yet, the user would see no reason for the poor performance. And, having explicitly defined the lifetime of a lock, it would be unwise to invalidate the least recently used one. Instead, POPM defines lock resource depletion to be a fatal error. In systems which need to survive such errors, near-depletion warnings might be possible.

4.0 Performance Expectations & Measurements

The key elements of the above concepts have been implemented. To evaluate the expected query performance, we have done a series of data transfer tests mimicking the patterns expected in data mining jobs. All tests were done on Fermilab's SP system, in which SCSI bus bandwidth is a bottleneck. The hardware is composed of 8 IBM SP-1 I/O nodes, each one with a single SCSI II fast interface which will sustain about 5.7 MBytes/second transfer rate, and 16 SP-2 compute nodes, all interconnected by TB-2 adapters to the high performance switch. With this hardware, using 64 kByte asynchronous messages, we have found that the MPL communications library can sustain 21 MBytes/second through a given node. Thus the aggregate bandwidth out of the I/O nodes is 168 MBytes/sec, while the aggregate sustained rate from the SCSI buses is only 45 MBytes/sec.

A perfectly balanced system would be able to read as much data as the compute nodes can process, but not more than that. This balance is affected by several factors besides the I/O subsystem: compute node CPU power, complexity of the query, number of stores scanned per query, etc. For our data mining queries, the optimum system would have one I/O node with 3 SCSI II fast interfaces per compute node. Thus, the current test system is clearly SCSI bandwidth limited and the test results reflect that.

Another important performance factor is the read-ahead algorithm: its depth will determine the number of disks and nodes working in parallel to exploit the striping of segments across storage units, and to hide access and delivery latency. The number of read-ahead segments is therefore one of the parameters varied in our tests.

Since aggregate switch bandwidth scales with system size and is not a limitation, a crucial measurement is the data bandwidth which can be concentrated into a single compute node, assuming that the consumer of that data is not compute bound. For a single query running on the system, the throughput will be a function of the communications performance, the segment prefetch depth, and the number of I/O nodes involved. Figure 7 shows the results:

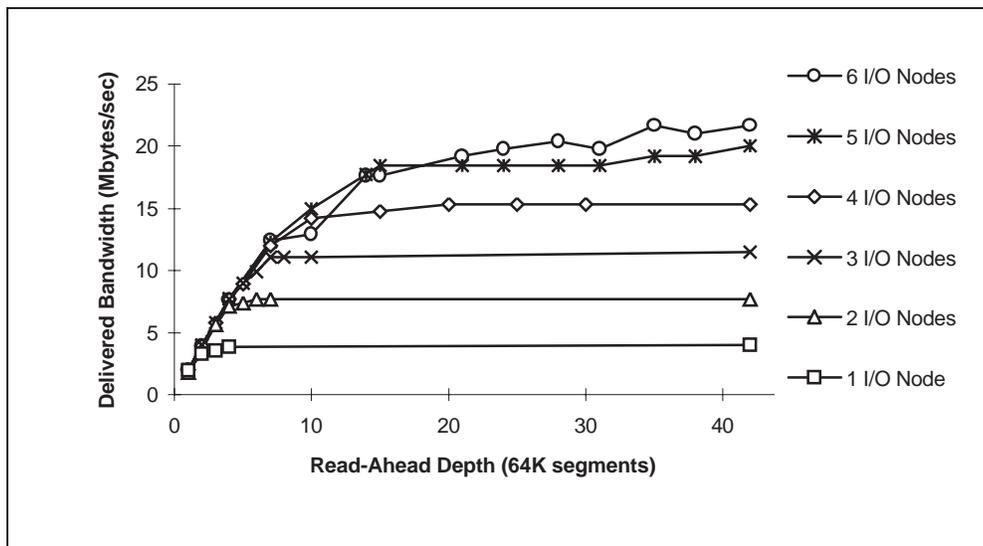


Figure 7. Delivered single query bandwidth

One of the most important properties of a distributed system is the scalability: the aggregate throughput should increase linearly with the addition of new hardware. To achieve this, a balance must be maintained among the capabilities of all crucial components - in our case disk bandwidth, switch bandwidth, and the computing power available. The net aggregate throughput cannot exceed the lowest of these limitations.

In our system, as discussed above, the number of SCSI adapters is the limiting factor. An aggregate throughput test was done using 7 I/O nodes each of which has a 5.7 Mbyte/sec sustained input rate; with perfect

parallelism, the maximum total throughput would be 39.9MB/s. The linearity seen in Figure 8 shows that, at least up to this performance level, POPM does a very good job of parallelizing the I/O.

Number of queries	Total throughput	Average query throughput
1	21.0	21.0
2	35.2	17.6
3	39.5	13.2

Figure 8. Throughput as a function of the number of queries.

These results show that the POPM concepts are accomplishing their design aims, to the extent permitted by the testbed hardware. Clearly, there is a need for better balance in Fermilab's hardware configuration. To this end, more disk I/O, and particularly SCSI bus bandwidth, is being added to the system. If most queries are not compute-bound, then the I/O:compute node ratio may also be adjusted. With these balance improvements, we should obtain up to 252 MBytes/sec of aggregate throughput, beyond which further improvements can only be achieved by scaling to a larger system.

5.0 Conclusion

Data mining is a way of life at Fermilab. It is required to get our job done. We have very large data processing needs and the volume of data to be processed doubles approximately every 18 months. As a result of this we are constantly pushing technology, searching for new and more efficient ways to solve our problems. Cost effective scalable, parallel data mining with commodity devices offers a very good solution to this specific problem.

There is an ever-increasing number of vendors who are making available competitive hardware suitable for this type application: Scalable systems with high I/O capabilities and high interconnection bandwidths. The major issue is in the software to effectively take advantage of these hardware configurations. POPM is a step in that direction, and is the best solution currently available for our needs. These needs are not unique in the marketplace; the POPM persistency management techniques are applicable in a variety of important contexts. POPM has been specifically designed to take advantage of other (newer) hardware when it becomes available and is not locked to the SP-2 architecture.

Beyond the efficient implementation of persistency across distributed I/O processors, POPM introduces some useful features for lightweight object managers in general. The syntax for dataset namespaces and the "LockedPptr" physical pointer concept are being incorporated into an improved Ptool API definition, so systems based on either persistency framework may easily be ported.

The positive results for performance behavior discussed above provides a proof-of-concept and verification that these techniques address the major limitations affecting query throughput in distributed-I/O systems. In particular, the ability to process data for a single query (on one compute node) at 21 Mbytes/sec provides encouraging validation of the concept of using such a system for high-performance complex analysis of event-oriented data.

6.0 References

- [1] R. L. Grossman, *Working With Object Stores of Events Using Ptool*, Laboratory for Advanced Computing Technical Report Number 94-8, University of Illinois at Chicago, 1994.
- [2] R.L. Grossman et al., *The Architecture of a Multi-level Object Store and its Application to the Analysis of High Energy Physics Data*, CERN Service De' Information Scientifique, Conference Proceedings, June 1994, pp66-97.
- [3] C. T. Day et al., *The PASS Project Architectural Model*, Computing in High Energy Physics, 1994, San Francisco, CA.
- [4] D. G. Feitelson et al., *Satisfying the I/O Requirement of Massively Parallel Supercomputers*, IBM T. J. Watson Research Report, RC-19008, July 15th, 1993.
- [5] K. Fidler et al., *The Computing Analysis Project - A High Performance Physics Analysis System*, to be published in the proceedings of Computers in High Energy Physics 1995, Rio De Janiero, Brazil.
- [6] M. C. Isely et al., *Design Notes for the Next Generation Persistent Object Manager for CAP*, Fermi National Accelerator Laboratory, Fermilab-TM-1934, 1995.