W-2 PSEIS, Blueprint for Parallel Processing

*RANDALL L. SELZLER
RSelzler@Data-Warp.com

Abstract: The Parallel Seismic Earth Imaging System (PSEIS[1]) is a software package that is envisioned as a successor to DDS[2]. It provides an alternative to MPI for parallel programming on distributed (and shared) memory machines. The objective is to make it easier to develop, maintain and execute parallel applications that are efficient and scalable. This is accomplished by integrating the technology into the I/O architecture and by tailoring it specifically to seismic processing. The new scheme provides a strategy for fault-tolerant processing, check-point restarts, persistent data storage, out-of-core solvers, data monitoring and parallel debugging.

Introduction: Large seismic data sets often need parallel programs to process them. MPI can work efficiently on distributed-memory machines, but is there something better? Some people, myself included, believe MPI is excessively difficult to learn, program and debug. If a better alternative were available for seismic processing, it could reduce software cost and facilitate the development of parallel programs by a wider audience.

The parallel programming technique proposed for PSEIS is suitable for distributed (and shared) memory machines. It should be efficient, scalable and easy to use for seismic processing. This is a big claim for such a challenging problem!

The basic concept (Figure 1) is to allow PSEIS data sets to use *RAM disk* (real memory) and allow multiple processes to bind offset ranges into their address space. The speed, capacity and functionality of RAM disk is similar to message passing, because all operations are memory based. This scheme leverages the Application Program Interface (API) used for PSEIS data sets. It minimizes the overall learning curve and integrates parallel processing into a seamless architecture. The open-read-write model for data transfer is also familiar to more programmers than the one used by MPI (mpirun, communicators, topologies, tags, messages, envelopes, send and receive).

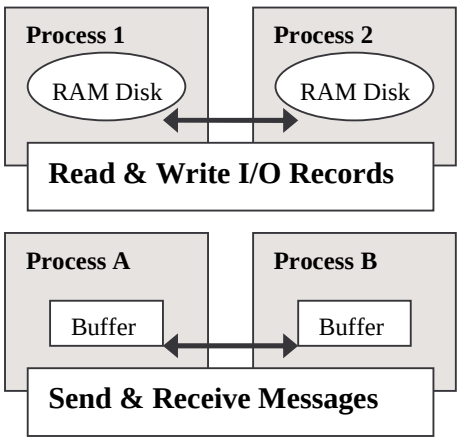

Figure 1: Similarity of MPI and I/O.

The PSEIS mechanism is designed specifically for seismic processing. The generality provided by MPI and byte I/O is desirable when dealing with diverse problems, however it is a liability when dealing with only one specific domain.

Parallel Processing: Parallel processing has three essential elements: *communication*, *synchronization*, and *process control*[3].

Communication: RAM data sets use an extended version of the open-read-write API. After a data set has been opened, a process can *bind* a range of offsets to a buffer in its address space. Other processes can open the same data set concurrently and bind offsets to their own buffers. Subsequent read and write operations within these ranges can be satisfied by a buffer (cache), regardless of where the request was initiated. This functionality is similar to one-sided sends and receives in MPI-2. PSEIS simply leverages a traditional I/O model, rather than imposing MPI's.

The bind mechanism provides important flexibility, convenience and efficiency.

A process can bind a range of offsets that correspond to a specific trace gather or specific elements of an array (like a finite-difference grid). The binding process has *direct access* to the buffer contents and other processes can access it via read and write. I/O requests can be serviced very efficiently, because the memory address for both the source and destination are known when requests are initiated.

N cooperating processes can each bind a *unique* range of offsets and either read (pull) or write (push) data as needed. *Point to point* communication patterns can be established, based upon the unique range associated with each process.

Alternatively, N cooperating processes can each bind the *same* range of offsets. In this case, one process can *broadcast* data to all other processes with one write to the shared range. PSEIS can efficiently and automatically update each cache by using a *propagation tree* to deliver the data.

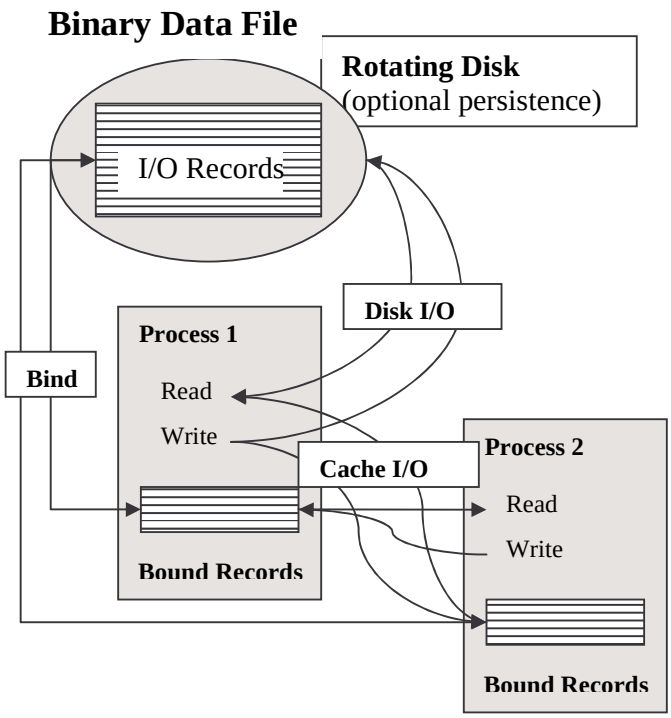

Figure 3: Binding and transferring I/O records.

Binding the same range is also useful when reading data. A *consumer's* read can be satisfied by the first *producer's* cache that becomes available. In this case, the producer's cache must be marked as unavailable (consumed), once it has been paired with a consumer. This mechanism can be used for load balancing and client-server support.

The bind mechanism can be used to implement *restart* functionality. For example, cache buffers can be explicitly *flushed* to persistent storage on rotating disk at *check- points* during processing. This requires one simple call to flush. If sync is also called, processing is protected from system crashes as well. The bind mechanism can automatically use the persistent storage (on rotating disk) to initialize cache buffers upon restart.

Bind can also be used to implement *fault-tolerant* functionality. For example, the broadcast mechanism can be used to preserve state information in independent processes and machines. This redundant state information can be used to automatically start an alternate process, if the primary one fails to respond.

A given process can open a data set and bind zero or more ranges of offsets to buffers. This can be used to *gather* and *scatter* data. For example, N cooperating processes can be responsible for producing (or consuming) every N_{th} trace within a data set. Another process can consume (or produce) a gather by reading (or writing) a contiguous sequence of traces. This can be efficiently implemented, because the scatter-gather pattern is specified once, in advance, and this knowledge can be used for optimization.

The binary format of disk and cache is specified the same way. More importantly, the format is encoded in a meta-data file (dictionary). The file can be read and decoded by other PSEIS

applications. MPI-2 lacks this functionality. The format of messages and I/O records is specified by a sequence of MPI calls. No support is provided for encoding and decoding format information in a meta-data file. Further more, MPI does not allow auxiliary information (units of measure, coordinates and annotation) to be associated with message fields. This makes it difficult for a suite of seismic applications to exchange data, if they use MPI. PSEIS solves this problem by keeping all meta-data about one data set in a dictionary.

The bind mechanism should not be confused with memory mapped I/O on Unix. They both use memory to shadow file contents, but the implementation and functionality is significantly different. Bind is implemented outside the OS using conventional file and socket I/O.

The Unix implementation may impose restrictions on the mapping. For example, the buffer may have to start on a page boundary. Traces and sub-arrays within applications are not generally aligned on pages. Unix maps use virtual memory tricks to identify and process pages that have been accessed or modified. Seismic applications need more explicit control of data transfer, because of efficiency and synchronization requirements. Finally, Unix semantics do not include synchronization of maps across distributed-memory machines.

Synchronization: When a data set is opened (or closed), a network connection is created (or destroyed) to all other processes that currently have the same data set open. These connections are used to transfer data and synchronize activity.

When a data set is opened, a unique *handle* is returned to the caller. Each handle has a *transaction queue*. This queue is used to coordinate activity associated with the data set.

Entries can be pushed onto the queue implicitly or explicitly. For example, a queue entry is *implicitly* created by open and close. Queue entries can also be implicitly pushed by read, write, bind, unbind, flush, sync, reset, lock and unlock. Processes can also *explicitly* push entries (simple messages) to other handles. This can be used to synchronize activity at intermediate levels, signal special conditions, implement barriers and reduce collective operations.

Processes can pop queue entries in FIFO order. Alternatively, they can automatically discard entries based upon their type. Processes must dispose of queue entries in a timely fashion, because it has a finite length. If the queue fills up, the pusher may be suspended, which can delay processing indefinitely.

The list of currently open handles and caches are used to initialize a new queue. Processes can monitor the queue and delay work until all parties have opened the data set (connected). A process can also terminate gracefully, when it sees others close the data (disconnect).

Data transfer and processing can be coordinated using intermediate or low-level synchronization. The required hand shake can be hardwired or negotiated at run time. These details are left to the application programmer and convenience routines that evolve to support them. The scheme is scalable, because communication and control can be decentralized.

Transaction queues can be used to coordinate access of data on rotating disk, not just cache buffers. Software and learning curves for parallel communication can be leveraged to tackle large problems that use out-of-core solutions. A combination of rotating disk and buffer cache can be used to provide a continuous range of solution options.

The transaction queue can also be used to debug parallel programs. Queue entries summarize the interaction between cooperating processes, which is essential knowledge. Entries, serial numbers and time stamps can be monitored in real time or preserved for subsequent analysis. Specialized tools can be created to assist with debugging.

Process Control: Processes can be created on multiple machines using any available mechanism. For example, they could be started manually by a user, launched by a batch

monitor or spawned by a cooperating process. A central authority, like mpirun, is not needed for communication.

Communication is established by opening the same data set. Data sets can be identified by name on a shared file system. Meta-data about an open data set is kept in its *connection* file. If it does not exist, then a new one is created by the first open. If it does exist, network connections are established and a new handle is added to the list of open ones. The open routine implicitly verifies and corrects the connection file. This sequence is reversed when a data set is closed. The last process removes the connection file.

Data sets can also be identified by a URL that references a daemon. The daemon can manage the meta-data and disk I/O for persistent storage. This functionality can be used to implement parallel disk I/O and WAN data access.

PSEIS optimizes communication by accumulating a map of all bound buffers. The system can quickly determine if a particular I/O record is cached, where it is and how to get it. Records may be bound and managed by applications or the *PSEIS I/O system* it self. For example, disk I/O that is cached at one handle can be accessed by another. In this case, the disk I/O system is leveraging parallel communication (which leverages disk I/O, recursively!).

Implementation: Implementing an efficient alternative to MPI is a daunting task, even if limited to a subset of MPI functionality. One approach is to use MPI itself for certain operations. It provides *lots* of flexibility! This could also leverage specialized drivers for high-performance networks. Functionality and licensing issues would have to be carefully investigated to determine whether MPI would be practical.

The PSEIS technology could be implemented using sockets. This might provide a simpler implementation to debug and maintain. It is unclear how much effort would be required to achieve a given level of efficiency.

Topology: An arbitrary topology can exist between cooperating processes and their data sets. The topology can change dynamically while processing. This functionality can be used for load balancing, fault recovery and monitoring purposes. For example, an interactive monitor could open an intermediate data set, even if it only resides in cache. The monitor could capture and display information for QC purposes, without disturbing the nominal processing and flow of data.

Conclusion: Data handling is fundamental. Get it right and other issues will fall into place!

Acknowledgment: Special thanks are due Jerry Ehlers, Joe Wade and Kyoung-Jin Lee for valuable input on this and related projects. Thanks are also due BP America for releasing the DDS software and for permission to publish this paper.

References:

[1] PSEIS http://PSEIS.org
[2] DDS http://FreeUSP.org/DDS/index.html
[3] Timothy Mattson, Beverly Sanders and Berna Massingill. “Patterns for Parallel Programming”. Addison-Wesley, 2004, ISBN: 0321228111.