Efficient Implementation of the MPI-IO System for Distributed Memory Multiprocessors

Maciej Brodowicz
University of Houston
Houston, TX, 77204
maciek@hpc.uh.edu

Expected graduation date: Spring 1998

My research concentrates on efficient implementation of parallel file systems. Its main focus are MPI-IO systems for distributed memory multiprocessors. The target platform for the developed code is the NEC Cenju-3 supercomputer, however the sources are directly portable to any homogeneous MPP running the Mach 3.0 microkernel. An important feature of my implementation is that I/O nodes of the underlying architecture need only the ability to transfer *blocks* of data between local disk and memory plus few standard file operations (`open`, `close`, `unlink`, `sync`). Thus, the developed MPI-IO system is designed to run not only on machines supporting standard UNIX file system, but also on ones equipped with less sophisticated interfaces (e.g. memory mapped files).

Unlike many common MPI-IO implementations providing only library support for MPI-IO functionality ([Thakur et al.], [Fineberg et al.]), this implementation consists of two components: library and server. The library (compliant with MPI-IO draft version 0.5 [MPI-IO][1]) is linked to the client application. It provides wrappers for MPI-IO functions and performs the first stage of request processing before submitting them to the server. This minimizes the load on the server (as the requests, which can be completely satisfied within library are never passed on to the server). Traffic reduction can also be achieved; a single remote procedure call to the server can in effect carry multiple data requests (especially if fragmented MPI datatypes are used). The servers' task is to relay file blocks from the remote to the requesting nodes and to keep track of them in a coherent manner. The blocks are cached exclusively at the server tasks, which simplifies consistency algorithms (no caching at the clients). Servers also coordinate accesses to the parallel files, so that, for example, deletion of a file requested by one of the applications is delayed until there are no accessors for that file, hence minimizing possible damage.

The current version of the MPI-IO library supports the full set of data transfer functions (including accesses with explicit, individual and shared pointers as well as their non-blocking counterparts). The library also includes environment initialization and termination calls, as well as file `open`, `close`, `delete`, `control`, `sync` and `seek` operations. Asynchronous request processing is achieved with set of `test`, `wait` and `request_free` functions using native MPI-IO representations. The library takes advantage of file hints (file `info`), which are used to define custom striping of file data across the disks. They also inform the system about anticipated access patterns.

The servers implement the parallel file system. In order to maximize the performance, a number of improvements (some suggested by theoretical research papers) was incorporated to the code:

Alternative access modes. Traditionally, coherency algorithms designate a single file block as the minimal consistency unit. While our server can operate in block-based access (multiple readers, single writer mode), we also investigate *direct update propagation*. In this mode, small amounts of data to be read or written are sent directly from (to) the block owner to (from) the requester, conserving the network bandwidth. Moreover, if all applications accessing the file conform to that mode, the target data blocks in most cases may be found in the server's cache, saving additional message latency to forward the request to the current block owner.

[1] The reason for not choosing the I/O chapter of the MPI-2 [Message-Passing Interface Forum] is that there exist no freely available implementations of that standard. Certain general features of MPI-2 are required to implement some functionality of MPI-IO, e.g. non-blocking requests.

For writes, an additional *write buffering* mode may be enabled, which explicitly trades atomicity control for improved write performance. The server accumulates data in especially marked, dynamically adjustable cache areas, without redistributing them to the block owning nodes. Since only minimal presorting is applied to keep track of data inserted to the *write buffer*, writes are performed almost with full memory bandwidth until the buffer becomes full. This event, or explicit request triggers the data propagation to the owning nodes, allowing for efficient use of the network bandwidth.

All above mentioned modes operate under the same, unified coherency scheme.

Cooperative caching. The server incorporates support for the idea first presented in [Dahlin et al.], which imposes collective management of all distributed caches in the system. The effects of discarding a block from cache are minimal, if we know that its copy is still cached elsewhere. At the same time ejection of *singlet* blocks is delayed as much as possible. Our server implements a distributed cooperative directory, which at the same time plays a role similar to hash queues in UNIX file systems (efficient block addressing). The replacement scheme is *N-chance forwarding*, which performed best in simulations mentioned in the original paper.

Low-level block presorting. Ordering blocks before performing a low level disk access is one of crucial ideas of David Kotz's *disk-directed I/O*. Currently, we entertain the idea of applying it automatically, without need for explicit synchronization of collective accesses to the disk. The server maintains two queues for urgent accesses and prefetch requests; all block accesses are sorted according to the block number. Of course, a starvation avoidance mechanism for out-of-order requests is provided. The requests are extracted from the queues by a single-threaded block I/O manager, with priority given to the urgent queue.

Tuned message passing. The efficiency of message passing in parallel I/O context has received much attention ([Chen et al.], [Cypher et al.], [Foster et al.]). This related work, however, is not always directly applicable to general parallel file servers due to overly restrictive assumptions (e.g. explicit synchronicity) or neglection of secondary effects (e.g. the universal assumption of linear dependence between message latency vs. message size). Our current experiments help to estimate message sizes and request merging factors for optimal performance.

REFERENCES

Chen, Y., Winslett, M., Seamons, K. E., Kuo, S., Cho, Y., and Subramaniam, M. 1996. Scalable message passing in Panda. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*. ACM Press, 109–121.

Cypher, R., Ho, A., Konstantinidou, S., and Messina, P. 1996. A quantitative study of parallel scientific applications with explicit communication. *Journal of Supercomputing 10,* 1 (March), 5–24.

Dahlin, M. D., Wang, R. Y., Anderson, T. E., and Patterson, D. A. 1994. Cooperative caching: using remote client memory to improve file system performance. *Proceedings of the First Symposium on Operating Systems Design and Implementation.*

Fineberg, S. A., Wong, P., Nitzberg, B., and Kuszmaul, C. 1996. PMPIO— a portable implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation.* IEEE Computer Society Press, 188–195.

Foster, I., Kohr, Jr., D., Krishnaiyer, R., and Mogill, J. 1997. Remote I/O: Fast access to distant storage. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems.* ACM Press, San Jose, CA, 14–25.

Message-Passing Interface Forum. 1997. *MPI-2.0: Extensions to the Message-Passing Interface.* MPI Forum, Chapter 9.

MPI-IO 1996. MPI-IO: a parallel file I/O interface for MPI. The MPI-IO Committee. Version 0.5. Available at http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps.

Thakur, R., Lusk, E., and Gropp, W. 1997. User's guide for ROMIO, a high-performance, portable MPI-IO implementation. Tech. Rep. ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory. October.