Multi-Threaded I/O (MTIO)

Alok Choudhary*  Ian Foster†  Sachin More‡  Ming Q. Xu§

Abstract

Rapid improvements in processor performance over the past decade have out-paced performance upgrades in I/O devices. The unrelenting pace of processor improvements has converted many traditionally compute-constrained tasks to ones dominated by I/O. Recent research efforts have been focused on optimizing file-server software. Some scalable I/O libraries developed provide a rich interface to help user make good choices on the data placement and data management policies. However many such libraries make a file server running on a dedicated processor and the processor would sit idle while the computing processors perform application processing. In this paper we present a multi-threaded approach to allow a user application to fully utilize all the computing resources by co-locating compute and I/O components. The Multi-threaded I/O (MTIO) library provides a UNIX-like file system calls that allow user to efficiently access files in a parallel computing environment. It is implemented on top of Nexus library and exploits the threading capability to perform I/O and it supports both task and data parallel paradigms effectively. It not only provides synchronous I/O but also the asynchronous I/O. Performance results demonstrate the effectiveness of the MTIO asynchronous I/O utility in overlapping the computation and I/O time in a physical processor thereby achieving better performances.

1 Introduction

Many scientific applications which have extensive I/O demands, together with massive computational requirements are now being implemented on parallel computers. In both multiprocessors and distributed systems, it has become increasingly obvious that the performance of many parallel applications is limited by their ability to move data between main memory and disk. This limitation arises when reading large input data sets, writing checkpoint and output data sets, and manipulating data sets too large to fit in main memory. The ultimate solutions to this problem require a coordinated effort at all levels, including architecture, algorithms, compilers, run-time libraries, and operating systems.

Operationally, an application can be viewed of as having two components, namely, compute and I/O. A compute component performs mostly application processing and the

*Dept. of Electrical and Computer Engineering, Syracuse University
†Argonne National Laboratory
‡Dept. of Electrical and Computer Engineering, Syracuse University
§Argonne National Laboratory
I/O component performs file-system processing. This makes the client-server model most natural for building scalable I/O libraries. As a matter of fact, the client server approach opens up avenues for I/O optimizing, especially in the context of collective I/O. The I/O requests from the applications many times requires lots of seeking and this is not really I/O friendly. However, instead of satisfying I/O requests on per demand basis, I/O servers can exchange I/O requests among themselves and coalescing I/O requests into fewer and larger I/O operations and thereby significantly reducing the I/O cost. Examples of such I/O libraries include Jovian [BBS+94], Passion [CBH+94] and Disk-directed I/O [Kot55]. However, many such I/O libraries uses the traditional heavy-weight UNIX processes to model both compute and I/O tasks. The major drawback of such an approach is two-fold:

\textit{lack of processor utilization} : some message passing libraries that these parallel libraries built on only allows one process per physical processor. So processors dedicated for I/O operations would remain idle when the compute processes are running with application processing.

\textit{overhead of context switching} : even it is possible to have both compute and I/O processes on the same physical processor, the context switching time would unfortunately offset the gain in I/O time.

\textit{communication cost} : inter-process communication normally costs more than sharing memory.

It is clear that a low cost mechanism is needed to co-locate both compute and I/O tasks on the same physical processor to maximize the utilization of parallel computing resources as well as reducing the overhead of context switching.

In this paper we describe a multi-threading technique for building parallel I/O libraries. Multi-threading is a well-known technique for masking communication latencies in parallel programs. The basic idea is to structure a program as a set of lightweight threads that are able to execute concurrently. When one thread is waiting for data from another processor, other threads can continue executing. This improves performance if the overhead associated with the use of multiple threads is less than the communication latency masked by computation.

The structure of the paper is as follows. §2 give a brief overview of MTIO system model and §3 the MTIO I/O request types and modes. Then, in §4, we present the implementation aspect o of the MTIO library and §5 describes the MTIO programming interface Performance results are presented in 6.

2 MTIO System Model

2.1 Client Server Model

Our MTIO design is based on a client/server model with a fundamentally different underlying mechanism. The system model is shown in figure 1. The model consists of a set processors and a file system. Each processor has two threads of control running on it,
namely, the compute thread and the I/O thread. The clients are the compute thread (CT) CT issues requests to its dedicated I/O thread (IOT) that manage logical input/output devices. The IOTs provide parallel file service to the application program.

CT In an SPMD computation, a client consists of an instance of the user application code. The compute thread, as the name suggests, is used for support computation the application code performs. When the compute thread needs to do I/O, it sends a request to the I/O thread. The I/O thread then services the request and sends back results (if any) to the compute thread. Though a compute thread on one processor never talks to either compute thread or I/O threads on the other processors for doing I/O, an I/O thread may talk to other I/O threads to satisfy a request.

IOT The I/O thread on the other hand is used exclusively for doing I/O. It provides an abstraction of an I/O device on a parallel computer. The IOTs are the ultimate resolvers of requests; all physical I/O flows through the IOTs. Each IOT consists of a cache of file data, storage for metadata associated with open files, IOT message passing support and an underlying file system for storing data. The IOTs also provides a rich set of I/O quests and I/O accessing modes which will be described in §3.
2.2 Software Organization

The MTIO library build around several layers of software as shown in Figure 2. The MTIO layer provides functionalities such as thread management and MTIO file routines. It implements the MTIO model.

The MPI layer provides the inter-process communication which the compute thread uses to perform application specific message passing.

The Nexus layer provides the low level support for parallelism and communication. Nexus is a runtime system integrating lightweight threads and communication. Nexus supports multiple threads of control, dynamic processor acquisition, dynamic address space creation, a global memory model via interprocessor references, and asynchronous events. Nexus is intended primarily as a compiler target for languages supporting task-parallel and mixed data- and task-parallel execution. The Nexus interface is organized around five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests. The associated services provide direct support for light-weight threading, address space management, communication, and synchronization. A computation consists of a set of threads, each executing in an address space called a context. An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context. It can also generate asynchronous remote service requests, which invoke procedures in other contexts.

The nexus layer sits atop of the target system layer which provides various thread-support (e.g. Pthreads, DCE threads, C threads and Solaris threads) and communication protocols and libraries (e.g. TCP/IP, ATM, MPL, NX, PVM, MPI, ATM, SVR4 shared memory and fiber channel).

The file system layer provides the support for I/O operations. It provides various vendor file systems (Unix NFS, IBM PIOFS and Unitree and Paragon CFS). The MTIO is designed to be portable across various platforms that provide different flavors of parallel I/O systems.

3 MTIO I/O Requests and Modes

3.1 I/O Requests

The system supports three kinds of I/O requests, namely, individual, collective and independent.

Individual I/O This is the simplest kind of I/O provided. The I/O thread, upon receiving the request, simply invokes the corresponding system call to service the request. See figure 3.

Collective I/O This kind of I/O request can be given when all the compute threads want to do I/O. In this case, the I/O threads talk to each other and decide upon a strategy to perform the I/O. Then each I/O thread uses the system calls to do the task assigned to it. Note that an I/O thread may service I/O request given by compute thread on another
processor partially or fully. A data exchange phase is involved either before the I/O starts (for write requests) or after the I/O is over (for read requests). See figure 4.

**Independent I/O** Here the I/O thread contacts other I/O thread and together the I/O threads service the request. See figure 5.

The difference between individual I/O and independent I/O is that with individual I/O, an I/O request from a CT is only handled by its own IOT, i.e. there is no communication between the IOTs in order to satisfy the I/O requests; whereas with independent I/O, an IOT may ask other IOTs to fulfill part of the I/O request. The independent I/O has the potential of reducing I/O requests by coalescing multiple I/O requests to form a larger I/O request and thereby reducing the seek time. Collective I/O matches the SPMD paradigm well as the SPMD application exhibit regular I/O access patterns. All the processors are either all performing computations or all performing I/Os. Individual and independent I/O is best suited for the task-parallel applications where I/O accesses
are more dynamic than what the SPMD applications exhibit. By supporting the above three types of I/O requests, MTIO library has the potential to support both the task parallel and data-parallel type of applications effectively.

3.2 I/O Modes

MTIO also supports two types of I/O modes, namely, synchronous and asynchronous I/O operations for all three types of I/O requests (individual, collective and independent) described above.

Synchronous and Asynchronous I/O  Figure 6 depicts the operations that are performed in both synchronous and asynchronous I/O operations. As for synchronous I/O, shown in figure 6a, the compute thread sends a request to the I/O thread, then the compute thread suspends until the I/O thread complete the I/O and informs the compute thread that the I/O is done. As for the asynchronous I/O, shown in figure 6b, the compute thread sends a request to the I/O thread. I/O thread then immediately returns a wait descriptor to the compute thread. Compute thread can then check completion of the I/O request calling a wait routine with the wait descriptor. The I/O thread meanwhile goes ahead with the I/O operation. When the I/O operation is finished, wait routine in the compute thread returns with data.

4 Implementing MTIO Library

4.1 Thread management and communication

The base model of MTIO is one of a fixed number of MTIO processes, one per physical processor, created at startup time. The MTIO processes then starts by initializing the MPI resources and various MTIO internal data structures. Following after that, the MTIO processes created the compute thread with a user provided compute function and the MTIO processes then enters a main loop where it acts as a I/O thread.

As a compute thread (CT) and an I/O thread (IOT) share the same address space, the communication between the two is therefore performed by using the shared memory communication scheme. As MPI is not threadsafe so we can only use MPI either among the IOTs or among CTs. We decide to make the CTs use MPI as they probably require a full blown message passing library whereas all the IOTs requires are non-blocking send and blocking receive. We have implemented the message passing routines used between the IOTs using nexus remote procedure request.

4.2 I/O operations

When a CT invokes an I/O operation, it sends a I/O request to its designated IOT. It then yields the processor explicitly so that the IOT could get scheduled. After the IOT gets scheduled, it reads the I/O request and process the I/O request, which may or may not involve other threads depending on the type of the I/O request. For synchronous I/O
Figure 6: Synchronous and Asynchronous I/O
requests, the I/O thread will return the control to the CT only when the I/O operations have completed whereas for asynchronous requests, the I/O thread return a synchronization data structure to the CT and CT could proceed with its computation immediately before the I/O requests have completed. The latter scheme could be used to overlap the computation with communication thereby offset the cost of I/O.

4.3 Data Structures

The system maintains the following data structures:

1. **Communication Data Structures**: These data structures are used for:
   
   (a) I/O request from the compute thread to the I/O thread
   (b) Responses to the compute I/O request from the I/O thread
   (c) Exchange of information between I/O threads for the collective I/O
   (d) I/O request from a I/O thread to other I/O threads for doing independent I/O
   (e) Responses from I/O threads to a I/O thread about independent I/O requests

2. **File table**: Each I/O thread maintains a file table that contains information about currently open files. The information stored in this table includes:

   - System file descriptor for the file in this I/O thread
   - MTIO file descriptors for the file other I/O threads
   - Number of processors that are actively using this file

3. **The Wait Structure**: The wait structure is a monitor which contains a mutex and a condition variable. It can be used to synchronize two threads of control. It also has a integer which stores the return value from the I/O routine.

5 Interface

The user will write a function called CPMain which will contain the code for the compute thread. The prototype is:

```c
void CPMain(int argc, char **argv);
```

The main function will create other processor objects on other processes. On each processor two threads of control will be created, one for computation and one for doing I/O. The program will expect command line arguments as:

```bash
myprog <numnodes> <hostfile> [<user arg1> <user arg2> ...]
```
The main function will use the first two arguments for creating the processor objects on appropriate machines. The remaining arguments will be passed on to the CPMain function.

The I/O thread code will be in function IOPMain. The prototype is:

\[
\text{void IOPMain(int mynode, int numnodes);}\]

The I/O library will consist of the following routines (the routines interface is similar to the UNIX counterpart):

1. \textit{mtio\_open} : This routine opens a file. The prototype is:

\[
\text{int mtio\_open(char *filename, int access\_flags[, int mode]);}\]

It returns a non-negative integer file descriptor on success.

2. \textit{mtio\_close} : This routine closes an open file.

\[
\text{int mtio\_close(int fd);}\]

The routine returns zero on successful completion.

3. \textit{mtio\_read} : This is a collection of routines to read an open file.

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Type</th>
<th>Synch/Asynch</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtio_read_ID</td>
<td>Individual</td>
<td>synch</td>
</tr>
<tr>
<td>mtio_read_IDA</td>
<td>Individual</td>
<td>asynch</td>
</tr>
<tr>
<td>mtio_read_CO</td>
<td>Collective</td>
<td>synch</td>
</tr>
<tr>
<td>mtio_read_COA</td>
<td>Collective</td>
<td>asynch</td>
</tr>
<tr>
<td>mtio_read_IP</td>
<td>Independent</td>
<td>synch</td>
</tr>
<tr>
<td>mtio_read_IPA</td>
<td>Independent</td>
<td>asynch</td>
</tr>
</tbody>
</table>

Table 1:

The prototype for the synchronous routines is:

\[
\text{int mtio\_read(int fd, char *buffer, int buflen, off\_t offset);}\]
The routine returns number of bytes read.

The prototype for the asynchronous routines is:

\[
\text{MTIO\_WAIT } \ast \text{mtio\_read\_A}(\text{int } \text{fd}, \text{char } \ast \text{buffer}, \text{int } \text{ buflen}, \text{off\_t } \text{offset});
\]

The routines return a pointer to the MTIO\_WAIT structure.

4. \textit{mtio\_write}: This is a collection of routines to write an open file.

\[
\begin{array}{|c|c|c|}
\hline
\text{Routine Name} & \text{Type} & \text{Synch/Asynch} \\
\hline
\text{mtio\_write\_ID} & \text{Individual} & \text{synch} \\
\text{mtio\_write\_IDA} & \text{Individual} & \text{asynch} \\
\text{mtio\_write\_CO} & \text{Collective} & \text{synch} \\
\text{mtio\_write\_COA} & \text{Collective} & \text{asynch} \\
\text{mtio\_write\_IP} & \text{Independent} & \text{synch} \\
\text{mtio\_write\_IPA} & \text{Independent} & \text{asynch} \\
\hline
\end{array}
\]

Table 2:

The prototype for the synchronous routines is:

\[
\text{int } \text{mtio\_write}(\text{int } \text{fd}, \text{char } \ast \text{buffer}, \text{int } \text{ buflen}, \text{off\_t } \text{offset});
\]

The routine returns number of bytes written.

The prototype for the asynchronous routines is:

\[
\text{MTIO\_WAIT } \ast \text{mtio\_write\_A}(\text{int } \text{fd}, \text{char } \ast \text{buffer}, \text{int } \text{ buflen}, \text{off\_t } \text{offset});
\]

The routines return a pointer to the MTIO\_WAIT structure.

5. \textit{mtio\_wait}: These routines is used to wait on/check for completion of an asynchronous request.

\[
\text{int } \text{mtio\_wait}(\text{MTIO\_WAIT } \ast \text{wait\_ptr});
\]
This function blocks till the asynchronous request (indicated by wait_ptr) finishes.

```c
int mtoi_check(MTIO_WAIT *wait_ptr);
```

This functions checks to see if the asynchronous request(indicated by wait_ptr) has finished.

5.1 An Example

The following example uses a collective read to extract a 8MB file in parallel.

```c
#include F_SIZE 1024 * 1024 * 8
void CMain(int argc, char **argv)
{
    int fd;
    int mynode, numnodes
    char *buffer

    MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
    MPI_Comm_size(MPI_COMM_WORLD, &numnodes);
    buffer = (char *) malloc(F_SIZE/numnodes);
    if (buffer == NULL) {
        fprintf(stderr,
                "Fail to allocate memory for the I/O buffer\n");
    }
    fd = mtoi_open(filename, O_RDONLY, 0644);
    mtoi_read_fd(fd, buffer, F_SIZE/numnodes,
                  mynode * (F_SIZE/numnodes));
    mtoi_close(fd);
}
```

6 Performance Results

6.1 System Configuration

We have conducted our experiments on the Argonne SP2. It has 128 processors and they are subdivided into 8 processor racks. One processor in each processor rack is configured as the I/O processor.