ORNL/TM-12743

Engineering Physics and Mathematics Division

Mathematical Sciences Section

DONIO: DISTRIBUTED OBJECT NETWORK I/O LIBRARY

E.F. D'Azevedo
C.H. Romine

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Date Published: November 1994

Research supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Contents

List of Figures

DONIO: DISTRIBUTED OBJECT NETWORK I/O LIBRARY

E.F. D'Azevedo

C.H. Romine

Abstract

This report describes the use and implementation of `DONIO` (Distributed Object Network I/O), a library of routines that provide fast file I/O capabilities in the Intel iPSC/860 and Paragon distributed memory parallel environments. `DONIO` caches a copy of the file in memory distributed across all processors. Disk I/O routines (such as `read`, `write`, and `lseek`) are replaced by calls to `DONIO` routines, which translate these operations into message communication to update the cached data. Experiments on the Intel Paragon show that the cost of concurrent disk I/O using `DONIO` for large files can be 15–30 times smaller than using standard disk I/O.

1. Introduction

Multi-megabyte disk input/output operations are commonly a major bottleneck in large application codes on distributed memory parallel supercomputers. This report describes the first version of the Distributed Network I/O (`DONIO`) library routines, which provide fast parallel file input/output capabilities on Intel iPSC/860 and Intel Paragon supercomputers. `DONIO` dramatically reduces the disk I/O time on the Paragon. The disk I/O time in a groundwater modeling code on 32 processors of a Paragon on a 52,000 node problem took 24 seconds using `DONIO`, as compared to 1212 seconds using the Intel `pfs` and native `NX` I/O calls. On a larger 1.3Million node problem running on 256 processors of the Paragon, the I/O time using `DONIO` grew to only 160 seconds.

This new Network I/O library (`DONIO`) provides fast operations by cacheing a copy of the disk file in memory distributed across all processors. Disk I/O requests are then translated into message communication to exploit the high network bandwidth for moving data. Actual disk operations are performed in large blocks using a few I/O processors to take advantage of RAID 0 striping across multiple disks.

2. Distributed Object Network I/O Library (DONIO)

`DONIO` is designed to speed up the I/O for distributed-memory parallel applications where all processors open a large multi-megabyte shared file for simultaneous access. To access a shared file, each processor relocates its own private copy of the file pointer with `lseek`'s to specific places in the file and then performs input/output operations. (Simultaneous output to overlapping regions in a shared file is nondeterministic; therefore, we assume that output operations do not overlap among processors). Such file access patterns are common in finite element codes that are based on subdomain decomposition. For example, the data for material properties or boundary conditions are commonly stored in shared files. This arrangement provides flexibility in solving the same problem with varying

numbers or configurations of processors without rearranging the data files.

A disadvantage of large shared files is that the overhead induced by many processors attempting to access the disk file concurrently can be quite large. Machines like the Intel i860 and Paragon attempt to support simultaneous access through a special file system (CFS for the i860, PFS for the Paragon). Even with this support, the cost for concurrent access to the same file can significantly degrade the performance of a parallel program. The performance of the current generation of Intel's CFS and PFS file systems is hampered by strict adherence to the OSF/1 standard. This in effect serializes the I/O to prevent any anomalous behavior of the file system.

`DONIO` offers a UNIX-like interface consisting of the 'C' callable primitives `do_open`, `do_read`, `do_write`, `do_lseek`, `do_lsize`, `do_flush` and `do_close`, which are similar to the `open`, `cread`, `cwrite`, `lseek`, `lsize`, `flush` and `close` routines provided by the Intel `NX` operating system. A Fortran callable interface, (*e.g.*, `DOREAD` for `do_read`), is also provided. Section 4 describes the use of these `DONIO` primitives in more detail. Changing the names of the I/O subroutines called in an application program from the `NX` version to the `DONIO` version (leaving the parameters untouched) and then linking in the `DONIO` library is generally all that is required to use the package. *An important note:* `DONIO` operates only on UNIX compatible binary files, which may be incompatible with Fortran unformatted fixed-size record files.

`DONIO` uses the simple idea of caching the entire disk file into the memory on the multiprocessor. Each processor has a limited amount of memory, so the cached data must be distributed among all processors. `do_read` and `do_write` access the cached copy in the aggregate memory instead of the disk file. Actual disk operations in `DONIO` are performed only during `do_open` for read-only and read-write files, and `do_close` for read-write and write-only files. For simplicity, `DONIO` only provides support for read-only, read-write and write-only files, *e.g.*, files that have been opened with flags `O_RDONLY`, `O_RDWR` and `O_WRONLY`, respectively. The `O_CREAT` flag is required for opening files that do not yet exist. Unlike

the UNIX `open` call, opening a write-only file in `DONIO` will overwrite the file if it already exists.

Most parallel supercomputers support a high performance parallel disk partition where disk records are striped across multiple disk for fast access. On the Intel machines, disk requests are serviced by dedicated I/O processors. Any actual disk I/O that is performed by `DONIO` operates on large blocks of contiguous data using the available I/O processors to take full advantage of RAID 0 striping across multiple disks. Note that the largest file permitted in the file system provided with Intel OSF/1 using the standard routines is 2 gigabytes. A 2 gigabyte file can be comfortably stored in `DONIO` with 4 megabytes each on 512 processors.

The (emulated) shared memory support in `DONIO` is provided by the Distributed Object Library (`DOLIB`) [1]. `DOLIB` is a set of library routines that allow dynamic creation (and destruction) of large global shared arrays on distributed memory environments, where these arrays are stored as "Distributed Objects" across all processors. `DONIO` stores the cached disk file as a one-dimensional array in global shared memory. `DONIO` translates disk operations such as `do_read` and `do_write` into `DOLIB gather/scatter` operations on the global array. In section 3, we provide a quick overview of the capabilities provided by `DOLIB`. For further details on the implementation of distributed objects using `DOLIB`, we refer the reader to [1].

3. Distributed Object Library (DOLIB)

A key component in `DONIO` is the transparent access to disk blocks cached in globally shared memory that is provided by the distributed-object library (`DOLIB`). `DOLIB` enables all processors to operate directly on any part of a distributed global array through explicit calls to `gather` and `scatter`. Advantages of using `DOLIB` include: dynamic allocation and freeing of huge (gigabyte) distributed arrays, both C and Fortran callable interfaces, and the ability to mix shared-memory and message-passing programming models for ease of programming and optimal performance.

DOLIB views a large global array as composed of fixed size pages stored in a block wrapped fashion across all processors. These pages can be easily malloc'ed or free'ed. Currently, DOLIB is implemented using the IPX (Inter Process eXecution) [3] system developed at Brookhaven National Laboratory.[1] DOLIB (and IPX) relies heavily on a reliable interrupt mechanism provided by hrecv on Intel multiprocessors. If a processor makes a call to gather, DOLIB first determines where (on which other processors) the requested data reside. For example, suppose that processor A requires data residing on processors B and C. The gather causes processor A to send message requests that interrupt processors B and C from regular computation. These processors package the requested data and send reply messages back to Processor A. They then exit this "interrupt" mode and resume regular computation. A similar sequence of messages is generated in a scatter operation.

The important facility provided by Intel's hrecv primitive is that all such signals are caught. For example, if processor B in the example above receives another interrupt while processing the one from processor A, the new interrupt is queued and then processed before processor B returns to normal execution mode. In section 7, we discuss how to implement DOLIB in the absence of such reliable signal handling.

For more details on the use of DOLIB as a programming paradigm for distributed memory multiprocessors, we refer the reader to [1].

4. User Interface

The following pages provide details on the syntax and behavior of each of the DONIO primitives. They form the manual pages for the eight procedures.

[1] IPX is available by anonymous FTP from the site msg.das.bnl.gov under the directory /pub/ipx.

do_nio

do_nio *initializes the* DONIO *system.* do_nio must *be called prior to opening any files with* do_open. *In* C, do_nio *returns 0 on success, -1 on failure.*

Synopsis

```
int do_nio( int myid, int nproc )

subroutine donio( myid, nproc )
integer myid, nproc
```

Input parameters

myid	–	myid is the id number of the calling processor.
nproc	–	nproc is the total number of processors executing.

Discussion

do_nio initializes the DONIO network I/O library. do_nio sets up internal data structures and initializes the DOLIB and IPX subsystems. Calling do_nio is required before any other calls to DONIO routines. Failure to do so will result in an error.

do_open

`do_open` *returns a non-negative descriptor on success. On failure, it returns -1. An implicit global synchronization is performed.*

Synopsis

```
#include <sys/fcntl.h>
int do_open( char *path, int flags, int mode )

include 'fnx.h'
integer function doopen( path, flags, mode )
character*(*) path
integer flags, mode
```

Input parameters

`path` – `path` is a null-terminated string that contains the pathname of the file.

`flags` – `flags` contains the access flags. Currently three access modes are supported: `O_RDONLY=0` for read-only access, `O_WRONLY=1` for write only access, and `O_RDWR=2` for read-write access. The latter two modes can be combined with `O_CREAT=512` (*e.g.,* `(O_WRONLY | O_CREAT)=513` or `(O_RDWR | O_CREAT)=514`) if the file does not exist.

`mode` – `mode` is the file permission (see chmod(2)) used in creating the output file. `mode` is ignored if the file already exists.

Discussion

The routine emulates the UNIX `open` (see open(2) in the UNIX manual), which opens the named file specified by `path` for read-only, write-only or

read-write access, as specified by the `flags` argument, and returns a descriptor for that file. For write-only or read-write access, if the file does not exist, it is created with permission mode `mode` (see chmod(2)). Note that `do_open` differs from UNIX `open` if the write-only file already exists. In that case, the file is first truncated (see truncate(2)) to an empty file and then rewritten.

All processors must participate in the `do_open` call. An implicit global synchronization is performed.

For read-only and read-write access, the entire file is read into global shared memory. This shared memory is deallocated on `do_close`. `do_open` may fail due to lack of memory. For write-only or read-write access, `do_open` should be followed immediately by a call to `do_lsize` that estimates the size of the output file. See the manual page for `do_lsize` for further details.

A Fortran example of the use of `do_open` is given below:

```
c ---
c ---  mode is set to octal 666,
c ---  full read-write permission
c ---
       mode = 8*8*6 + 8*6 + 6
       rflags = 0
c ---
c ---  be sure path is null terminated
c ---
       path = '/pfs/infile' // char(0)
       fd = doopen( path, rflags, mode )
```

do_lsize

`do_lsize` *estimates the size of the write-only or read-write output file associated with file descriptor* `fd`. *Calling* `do_lsize` *is* required *after the file has been opened with* `do_open` *and prior to any call to* `do_write` *involving the file. In* `C`, `do_lsize` *returns* `nbytes` *on success. An implicit global synchronization is performed.*

Synopsis

```
int do_lsize( int fd, int nbytes )
```

```
subroutine dolsize( fd, nbytes )
integer fd, nbytes
```

Input parameters

`fd`	–	`fd` is the file descriptor obtained from `do_open`.
`nbytes`	–	`nbytes` is the estimated file size in bytes.

Discussion

`do_lsize` allocates the requested space before starting write operations. Underestimation of file size will lead to an *error* condition. Overestimation will lead to unnecessarily high memory use but the actual file generated on disk will be of correct/minimal size. `do_lsize` should be called immediately after `do_open` for write-only and read-write files. Calling `do_lsize` for files opened for read-only access results in an error.

All processors must participate in the `do_lsize`. An implicit global synchronization is performed.

do_lseek

`do_lseek` *sets the (local) seek pointer of the open file associated with the file descriptor and returns the new seek position.*

Synopsis

```
#include <unistd.h>
int do_lseek( int fd, int offset, int whence )

include 'fnx.h'
integer function dolseek( fd, offset, whence )
integer fd, offset, whence
```

Input parameters

`fd`	–	`fd` is the file descriptor obtained from `do_open`.
`offset`	–	`offset` is the offset in bytes.
`whence`	–	`whence` determines the computation with offset. `whence` is one of `SEEK_SET=0`, `SEEK_CUR=1` or `SEEK_END=2`.

Discussion

`do_lseek` sets the seek pointer associated with the open file specified by the descriptor `fd` according to the value supplied for `whence`. `whence` must be one of `SEEK_SET=0`, `SEEK_CUR=1`, `SEEK_END=2` defined in `<unistd.h>` (see lseek(2)).

If `whence` is `SEEK_SET`, the seek pointer is set to `offset` bytes. If `whence` is `SEEK_CUR`, the seek pointer is set to its current location plus `offset`. If `whence` is `SEEK_END`, the seek pointer is set to the size of the file plus `offset`.

`do_lseek( fd, 0, SEEK_END)` returns the size (in bytes) of the opened file associated with `fd`.

do_read

`do_read` *performs a read operation into the specified buffer. In* `C`*,* `do_read` *returns the number of bytes read.*

Synopsis

```
int do_read( int fd, void *buf, int nbytes )

subroutine doread( fd, buf, nbytes )
integer fd, buf(*), nbytes
```

Input parameters

`fd`	–	`fd` is the file descriptor obtained from `do_open`.
`buf`	–	`buf` is the buffer.
`nbytes`	–	`nbytes` is the number of bytes to be read.

Description

`do_read` attempts to read `nbytes` bytes of data from the file referenced by the descriptor `fd` into the buffer `buf` (see read(2)).

This is a synchronous call. The calling process waits (blocks) until the request is completed. Note that reading past the end of file causes an *error*. Calling `do_read` to read from a write-only file causes an *error*. The seek pointer is updated to point to the next byte in the file.

do_write

`do_write` *performs a write operation from the specified buffer. In* `C`*,* `do_write` *returns the number of bytes written.*

Synopsis

```
int do_write( int fd, void *buf, int nbytes )

subroutine dowrite( fd, buf, nbytes )
integer fd, buf(*), nbytes
```

Input parameters

`fd`	–	`fd` is the file descriptor obtained from `do_open`.
`buf`	–	`buf` is the buffer.
`nbytes`	–	`nbytes` is the number of bytes to be written.

Description

`do_write` attempts to write `nbytes` bytes of data to the file referenced by the descriptor `fd` from the buffer `buf` (see write(2)).

This is a synchronous call. The calling process waits (blocks) until the request is completed. Note that writing past the estimated size of file determined in `do_lsize` causes an *error*. Calling `do_write` before `do_lsize` has been called causes an error. Calling `do_write` to write to a read-only file causes an *error*. The seek pointer is updated to point to the next byte in the file.

do_flush

`do_flush` *forces* `DONIO` *to write the cached file associated with the given file descriptor to the disk. In* `C`*,* `do_flush` *returns 0 on success and -1 on failure. An implicit global synchronization is performed.*

Synopsis

```
int do_flush( int fd )

subroutine doflush( fd )
integer fd
```

Input parameters

`fd` – `fd` is the file descriptor obtained from `do_open`.

Discussion

`do_flush` forces an immediate write of the specified cached file to disk. `do_flush` is provided to support checkpointing, since in the event of a machine malfunction, all data written to the cached file will be lost. `DONIO` automatically keeps track of the largest byte addressed with `do_write`, so the disk file will have the correct size. However, unwritten bytes (*i.e., gaps*) in the file will contain garbage. Subsequent calls to `do_flush` with the same argument will overwrite the disk file, rather than appending to it. However, if no changes have been made to the cached file since the last call to `do_flush` no disk I/O will take place.

All processors must participate in the `do_flush` call. An implicit global synchronization is performed.

do_close

`do_close` *closes the file associated with the file descriptor and deallocates global shared resources.* `do_close` must *be called to ensure that any writes are saved to disk. In* `C`, `do_close` *returns 0 on success and -1 on failure. An implicit global synchronization is performed.*

Synopsis

```
int do_close( int fd )
```

```
subroutine doclose( fd )
integer fd
```

Input parameters

`fd` – `fd` is the file descriptor obtained from `do_open`.

Discussion

`do_close` deallocates the global shared resources used for cacheing the file data associated with the file descriptor `fd`. For write-only and read-write files, `do_close` first initiates the actual disk operations, if necessary, to write out the cached data to the disk file before resources are deallocated. (If no changes have been made to a read-write file, no disk I/O is performed).

Important note: Unlike the UNIX routines, no implicit `do_close` *calls are performed when the program terminates. Hence, if the user fails to call* `do_close` *for a given file, any writes to the file will be lost upon program termination!* All processors must participate in the `do_close` call. An implicit global synchronization is performed.

5. Implementation Details

DONIO is designed to emulate, in large part, the UNIX file I/O routines. In this section, we discuss several design decisions that lead to differences between them.

DONIO uses DOLIB to implement a cached disk file as a globally distributed array of bytes. DOLIB provides bounds checking on such global arrays; consequently, the size of an array in DOLIB (and hence the size of a file in DONIO) is determined at creation time. For read-only files, the size is determined from the disk file. For write-only or read-write files, DONIO requires the user to specify the eventual maximum file size by calling do_lsize. Attempting file access beyond the specified maximum results in a reference to a nonexistent DOLIB array element, which is flagged as an error.

In many applications, it may be difficult for the user to ascertain in advance the exact eventual size of the file being written. However, DONIO automatically keeps track of the highest address actually used. If the user overestimates the file size in do_lsize then the correct (exact) size file will be written to disk. Overestimating the file size means only that the (unused) extra allocated memory will be unavailable for the user's application.

In DONIO all processors must participate concurrently in do_open, do_lsize do_flush and do_close. The processors are synchronized when opening a shared file with do_open so that DOLIB can set up common data structures. Processors are synchronized when specifying the file size with do_lsize, which determines how much memory each should allocate. They are synchronized in do_flush and in do_close to ensure that there are no outstanding read/write requests.

When DONIO opens an existing disk file with write-only access, there is no guarantee that DONIO will have read permission on the file, so the current contents cannot be cached across the processors. For simplicity, we have decided to truncate the file to zero length. Existing files that are to be updated using DONIO should be opened with read-write access.

DONIO does not support an APPEND mode for file I/O. APPEND mode is most often used for writing out intermediate computational results such as results

for each time-step in a time-dependent calculation. If write-only mode is used instead, the eventual size of the file may be so large that it is impractical to allocate memory for it. We recommend that the user open separate files for each logically separate set of data, *e.g.*, a separate file for each time-step.

We have demonstrated that `DONIO` has significant advantages over the standard I/O routines for concurrent I/O. However, unlike the standard routines, actual disk I/O is performed during `do_open` and `do_close` depending on the file access flags. Hence, the cost of these routines may seem unusually high compared to the UNIX routines `open` and `close`.

6. Experimental Results

In this section we present a rough comparison of disk performance by `DONIO` versus native `NX` routines. The Fortran source code is included in the Appendix. The code is a contrived example that simulates the disk I/O common in finite element codes by performing multiple direct access `lseek`'s, `read`'s and `write`'s. This example generates the element-to-vertex list for a three dimensional $nnx \times nny \times nnz$ grid. The elements are assumed to be ordered with z-index varying fastest, then x then y. Elements along the vertical direction are grouped in buffer `mibuf` before writing, to obtain better disk performance. Note that the element-to-vertex list file is independent of the number of processors. The same file is later read again.

Since operating system patches and compiler upgrades are regularly applied to the 512-processor Paragon at Oak Ridge National Laboratory and `DONIO` is currently undergoing performance tuning, the performance numbers listed should be taken only as approximate and reflect only the current state of affairs.

Three problems were used for testing: a small $41 \times 41 \times 31$ (48,000 elements) problem, a medium $81 \times 81 \times 61$ (384,000 elements), and a large $121 \times 121 \times 91$ (1,296,000 elements) problem. Timings for `NX` native routines on the largest problem were over 1,000 seconds. These times were highly variable since the machine was not dedicated to our application, and hence they are not reported.

Table 6.1: DONIO routines on $41 \times 41 \times 31$ grid, file size is 1,536,000 bytes.

processor	wopen	write	wclose	ropen	read	rclose
4	0.55	2.45	3.04	2.30	14.67	0.01
8	0.58	1.39	3.06	2.35	8.19	0.01
16	0.59	0.85	2.86	2.69	3.70	0.01
32	0.60	0.55	3.13	2.36	1.81	0.01
64	0.62	0.37	2.64	2.50	1.10	0.02

Table 6.2: NX routines on $41 \times 41 \times 31$ grid, file size is 1,536,000 bytes.

processor	wopen	write	wclose	ropen	read	rclose
4	1.95	60.03	0.24	1.19	34.97	0.24
8	4.02	54.11	0.45	2.24	43.41	0.45
16	6.86	60.74	0.71	6.07	39.47	0.71
32	14.23	59.15	0.93	17.70	41.11	1.25
64	53.81	65.10	2.86	44.53	42.56	3.05

Tables 6.1–6.5 list the runtimes obtained from `dclock()`. `wopen` (`wclose`) denotes the time for opening (closing) a file for write-only access; similarly, `ropen` and `rclose` apply to read-only access. Note that time consuming actual disk operations are performed in `DONIO` during `wclose` and `ropen`. Only 4 I/O processors were used in `DONIO`, hence actual disk I/O time is largely insensitive to the total number of processors.

Figures 6.1–6.5 present graphical views of the results. Note that `read` and `write` times in `DONIO` decrease with the addition of more processors; since as more processors are used, fewer messages *per* processor are generated. On the other hand, `NX` disk operations are handled by 6 dedicated I/O processors. For a given problem the total number of disk requests is fixed, and hence I/O times do not decrease with more processors.

We see that on all test cases, *total* time for `DONIO` is over 15 times faster than using native `NX` routines.

Table 6.3: DONIO routines on $81 \times 81 \times 61$ grid, file size is 12,288,000 bytes.

processor	wopen	write	wclose	ropen	read	rclose
4	0.72	13.53	13.05	20.76	67.56	0.01
8	0.57	7.33	13.53	11.72	31.51	0.01
16	0.59	3.85	10.85	14.98	16.30	0.01
32	1.09	2.15	8.80	10.86	8.55	0.01
64	0.66	1.29	8.84	12.07	4.87	0.01

Table 6.4: NX routines on $81 \times 81 \times 61$ grid, file size is 12,288,000 bytes.

processor	wopen	write	wclose	ropen	read	rclose
4	1.71	241.02	0.25	1.75	182.80	0.18
8	4.24	237.12	0.46	3.31	162.70	0.46
16	9.52	231.62	0.78	9.54	179.01	0.74
32	16.95	247.22	1.32	23.92	185.91	1.05
64	51.18	239.68	3.12	45.47	182.56	2.79

Table 6.5: DONIO routines on $121 \times 121 \times 91$ grid, file size is 41,472,000 bytes.

processor	wopen	write	wclose	ropen	read	rclose
8	0.95	20.69	46.13	56.91	77.19	0.01
16	0.81	10.79	41.44	47.00	36.32	0.01
32	0.61	5.65	36.97	40.80	21.16	0.01
64	0.64	3.03	41.64	42.67	11.57	0.01
128	0.65	1.80	33.05	39.22	6.90	0.02

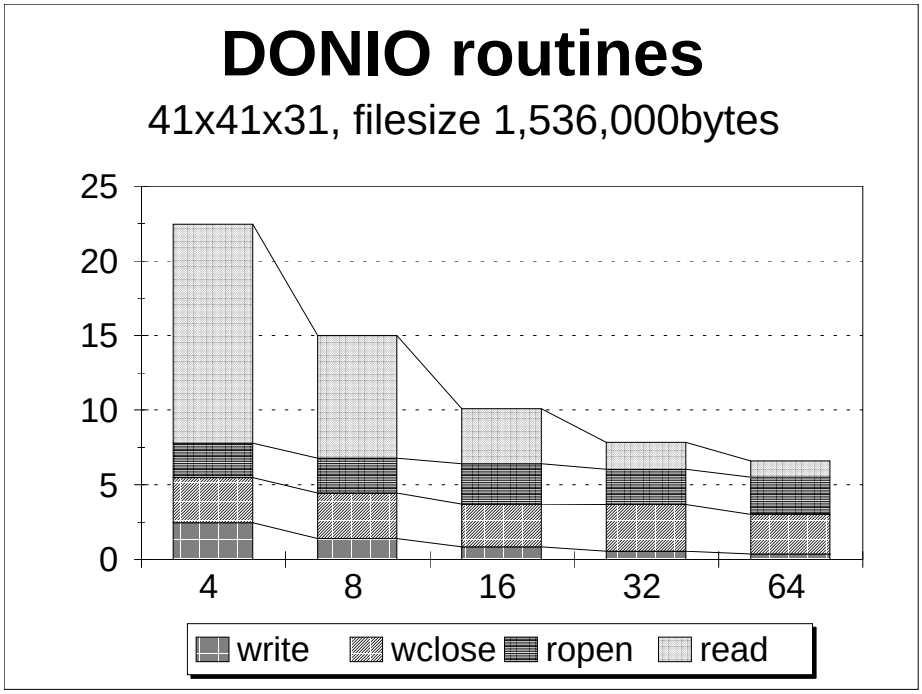

Figure 6.1: `DONIO` on small problem.

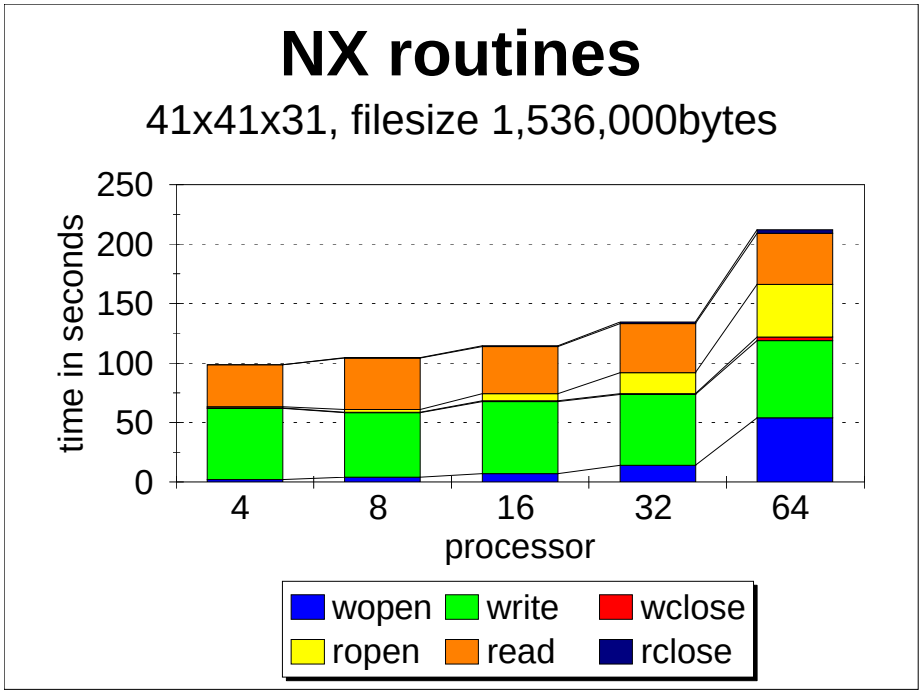

Figure 6.2: `NX` on small problem.

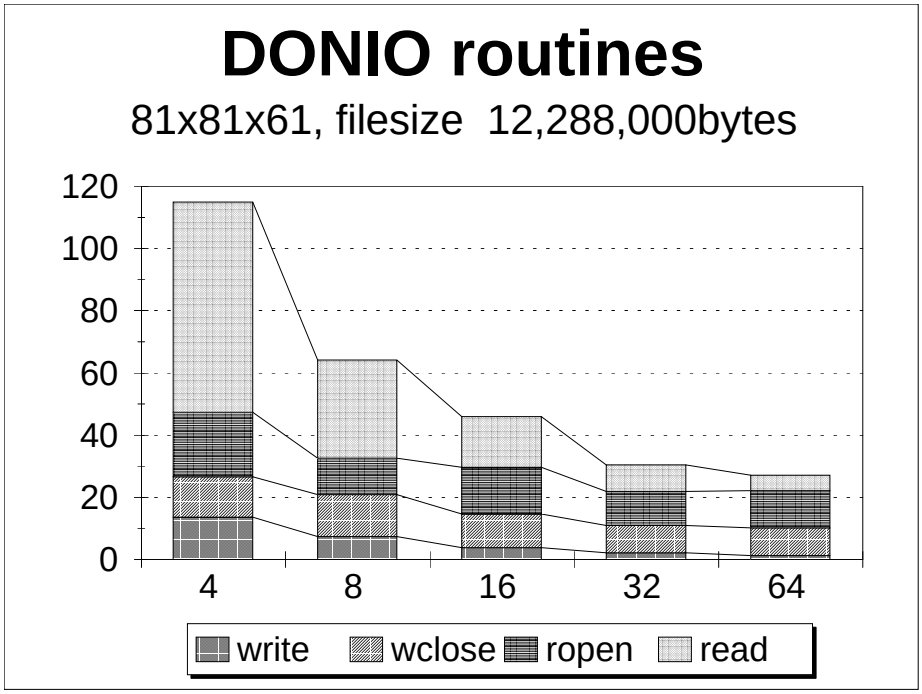

Figure 6.3: `DONIO` on medium problem.

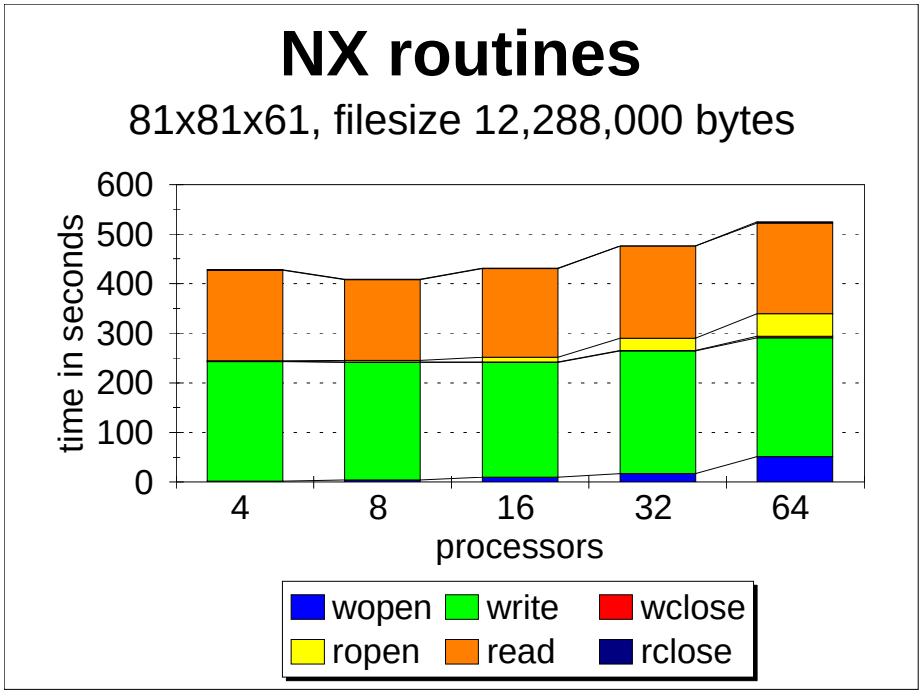

Figure 6.4: `NX` on medium problem.

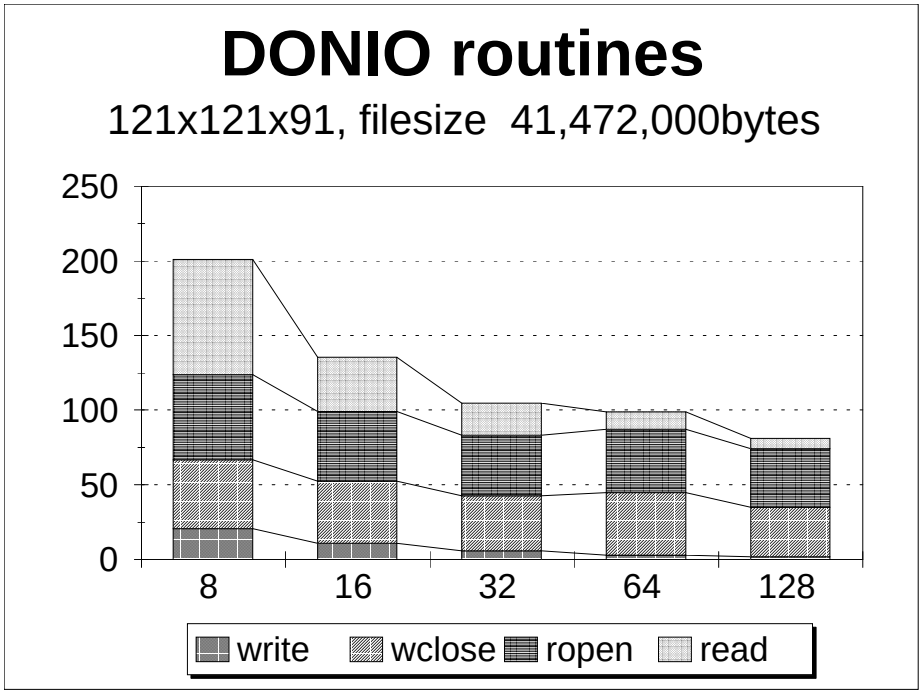

Figure 6.5: `DONIO` on large problem.

7. PVM Implementation

PVM, or Parallel Virtual Machine, is currently one of the most widely used message-passing paradigms [2]. A well-written distributed-memory parallel code using PVM will run on a wide class of machines, from supercomputers to heterogeneous collections of workstations. We intend to take advantage of this portability by creating versions of DONIO and DOLIB that use PVM for message-passing.

Unfortunately, the interrupt capabilities of PVM are limited to unreliable UNIX signals. There is no guarantee that signals will not be lost. Under PVM, we propose to have DOLIB periodically *poll* for pending messages. PVM's signal handler will be used only to induce a process to check its message queue.

For further discussion of how DOLIB should be implemented under PVM, the reader is referred to [1].

8. Summary

We have described DONIO, a fast file I/O emulation library for the Intel iPSC and Paragon distributed memory multiprocessors. DONIO provides an easy to use interface that, with minimal change to the source of an iPSC/860 or Paragon parallel program, can speed up file I/O by a factor of 15 to 30 times. DONIO creates a copy of the disk file in the aggregate memory of the multiprocessor. Disk I/O operations are replaced with the matching DONIO routines, which effect memory updates to this copy. DONIO relies on the underlying library DOLIB, which supports the creation, use and destruction of globally shared arrays in distributed-memory environments.

9. Obtaining the Software

To obtain the source code for DONIO the reader should send email to the authors: efdazedo@msr.epm.ornl.gov or rominech@ornl.gov.

Acknowledgements

The authors would like to express appreciation to Bob Marr, Ron Peierls and Joe Pasciak for the `IPX` package, which simplified the development of `DONIO`. We also thank David Walker and Pat Worley for suggesting improvements both to `DONIO` and to this report.

10. Appendix

In this appendix, we list the Fortran source code used in comparing the performance of DONIO and NX disk operations. Note that either DONIO or NX routines can be selected by a flag at compile time.

```
        program  ex1
c---
c---    a simple example to illustrate the use of DONIO
c---
        include 'fnx.h'
#ifdef USE_NX
c---
c--- note: fd is defined as a constant unit number
c---
        integer fd
        parameter(fd=16)

#define IOINIT(myid,nproc)
#define LSEEK lseek
#define ROPEN(fd, filename) open(fd,file=filename,form='unformatted')
#define WOPEN(fd, filename) open(fd,file=filename,form='unformatted')
#define LSIZE(fd, newsize)  ierr = lsize( fd, newsize, SIZE_SET )
#define CREAD(fd, ibuffer,nbytes)  call cread(fd,ibuffer,nbytes)
#define CWRITE(fd, ibuffer, nbytes)  call cwrite(fd, ibuffer, nbytes )
#define CCLOSE(fd) close( fd )
#define GSYNC gsync

#else

        integer rflags,wflags,mode
        parameter(rflags=0,wflags=(512+1),mode=(8*8*6+8*6+6))
        integer doopen, doread, dowrite, dolseek
        external doopen, doread, dowrite, dolseek
        external doclose,dolsize

c---
c---    note: fd is declared as a variable
c---
        integer fd

#define IOINIT(myid,nproc)  call donio(myid,nproc)
#define LSEEK dolseek
```

```
#define ROPEN( fd, filename) fd = doopen( filename, rflags,mode)
#define WOPEN( fd, filename) fd = doopen( filename, wflags,mode)
#define LSIZE( fd, newsize ) call dolsize( fd, newsize )
#define CREAD(fd, ibuffer,nbytes)  call doread(fd, ibuffer, nbytes )
#define CWRITE(fd, ibuffer, nbytes) call dowrite( fd, ibuffer, nbytes )
#define CCLOSE( fd ) call doclose(fd)
#define GSYNC dogsync

#endif

        integer indev,outdev,sizeint,nvertex,maxnez
        parameter(indev=5,outdev=6,sizeint=4,nvertex=8,maxnez=1024)


        double precision tstart,tend
        character*80 filename
        integer i, ix,iy,iz,  nnx,nny,nnz,  nex,ney,nez
        integer mbuf(nvertex,maxnez)
        integer nbytes,totalbytes,   myid,nproc
        integer mi,i,ierr,offset,  iwork
        logical ismine
c---
c---    8 vertices of an hexahedral brick element
c---
        integer dx(nvertex),dy(nvertex),dz(nvertex)
        data dx /0,1,1,0,   0,1,1,0/
        data dy /0,0,1,1,   0,0,1,1/
        data dz /0,0,0,0,   1,1,1,1/


        integer ijk2mi,ijk2ni
        ijk2mi(ix,iy,iz,nex,ney,nez) = iz+(ix-1)*nez+(iy-1)*nez*nex
        ijk2ni(ix,iy,iz,nnx,nny,nnz) = iz+(ix-1)*nnz+(iy-1)*nnz*nnx
c---
c---    code begins
c---

        myid = mynode()
        nproc = numnodes()
        IOINIT( myid, nproc )

        nnx = 0
        nny = 0
        nnz = 0
        if (myid .eq. 0) then
```

```
                write(outdev,*) 'enter nnx,nny,nnz '
                read(indev,*) nnx,nny,nnz
                write(outdev,*) 'nproc, nnx,nny,nnz ', nproc,nnx,nny,nnz
        endif

        call gisum(nnx,1,iwork)
        call gisum(nny,1,iwork)
        call gisum(nnz,1,iwork)

        nex = nnx - 1
        ney = nny - 1
        nez = nnz - 1
        totalbytes = (nex*ney*nez)*nvertex*sizeint

        call GSYNC()
        tstart = dclock()
#ifdef USE_NX
        filename = '/pfs/nxex.bin'
#else
c---
c---    IMPORTANT NOTE: string MUST be null terminated
c---
        filename = '/pfs/ex.bin' // char(0)
#endif
        WOPEN( fd, filename )
c---
c---    ESSENTIAL to call dolsize
c---
        LSIZE( fd, totalbytes )
        call GSYNC()
        tend = dclock()
        if (myid .eq. 0) then
            write(outdev,*) ' open/lsize takes ', tend-tstart,' sec'
            write(outdev,*) ' total file size is ', totalbytes,' bytes'
        endif
c
        nbytes = nvertex*sizeint
        call GSYNC()
        tstart = dclock()
        do ix=1,nex
        do iy=1,ney

           ismine = (mod( ix+(iy-1)*nex, nproc) .eq. myid )

           if (ismine) then
```

```
            do iz=1,nez
                do i=1,nvertex
                 mbuf(i,iz)=ijk2ni(ix+dx(i),iy+dy(i),iz+dz(i),nnx,nny,nnz)
                enddo
            enddo

            mi = ijk2mi( ix,iy,1,   nex,ney,nez)
            offset = (mi-1)*nvertex*sizeint
            ierr =  LSEEK( fd, offset, SEEK_SET )
            nbytes = nez*nvertex*sizeint
            CWRITE( fd, mbuf(1,1), nbytes )
           endif
         enddo
         enddo

         call GSYNC()
         tend = dclock()
         if (myid .eq. 0) then
             write(outdev,*) ' write takes ', tend - tstart,' sec'
         endif

         call GSYNC()
         tstart = dclock()
         CCLOSE( fd )
         call GSYNC()
         tend = dclock()
         if (myid .eq. 0) then
             write(outdev,*)' close for write takes ',tend-tstart,' sec'
         endif

c ---
c --- read the element list back
c ---
         call GSYNC()
         tstart = dclock()
         ROPEN( fd, filename )
         call GSYNC()
         tend = dclock()
         if (myid .eq. 0) then
             write(outdev,*)' open for read takes ', tend-tstart,' sec'
         endif


         nbytes = nvertex*sizeint
         call GSYNC()
```

```
tstart = dclock()

do ix=1,nex
do iy=1,ney
   ismine = (mod( ix+(iy-1)*nex, nproc) .eq. myid )
   if (ismine) then
        mi = ijk2mi( ix,iy, 1,    nex,ney,nez)
        offset = (mi-1)*nvertex*sizeint
        ierr = LSEEK( fd, offset, SEEK_SET )

        nbytes = nez*nvertex*sizeint
        CREAD( fd, mbuf(1,1), nbytes )
   endif
enddo
enddo

call GSYNC()
tend = dclock()
if (myid .eq. 0) then
    write(outdev,*) ' all reads take ',tend-tstart,' sec'
endif

call GSYNC()
tstart = dclock()
CCLOSE( fd )
call GSYNC()
tend =  dclock()
if (myid .eq. 0) then
    write(outdev,*) ' close for read takes ', tend-tstart,' sec'
endif

stop
end
```

11. References

[1] E. F. D'Azevedo and C. H. Romine, *DOLIB: Distributed object library*, Tech. Report ORNL/TM-12744, Oak Ridge National Laboratory, 1994.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 user's guide and reference manual*, Tech. Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.

[3] B. Marr, R. Peierls, and J. Pasciak, *IPX – Preemptive remote procedure execution for concurrent applications*, Tech. Report, Brookhaven National Laboratory, 1994.