

Computer Science and Mathematics Division

Mathematical Sciences Section

**EDONIO: EXTENDED 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: March 1995

Research supported by the Applied Mathematical Sciences  
subprogram of the Office of Energy Research, U.S. Depart-  
ment 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

1	Introduction . . . . .	1
2	Extended Distributed Object Network I/O Library . . . . .	1
3	User Interface . . . . .	3
4	Implementation Details . . . . .	18
5	Experimental Results . . . . .	21
6	Summary . . . . .	22
7	Obtaining the Software . . . . .	24
8	Appendix . . . . .	25
9	References . . . . .	32

# EDONIO: EXTENDED DISTRIBUTED OBJECT NETWORK I/O LIBRARY

E.F. D'Azevedo

C.H. Romine

## Abstract

This report describes **EDONIO** (Extended Distributed Object Network I/O), an enhanced version of **DONIO** (Distributed Object Network I/O Library) optimized for the Intel Paragon Systems using the new **M\_ASYNC** access mode. **DONIO** provided fast file I/O capabilities in the Intel iPSC/860 and Paragon distributed memory parallel environments by caching a copy of the *entire* file in memory distributed across all processors. **EDONIO** is more memory efficient by caching only a subset of the disk file at a time. **DONIO** was restricted by the high memory requirements and use of 32-bit integer indexing to handle files no larger than 2Gigabytes. **EDONIO** overcomes this barrier by using the extended integer library routines provided by Intel's **NX** operating system.

For certain applications, **EDONIO** may show a ten-fold improvement in performance over the native **NX** I/O routines.

## 1. Introduction

Multi-megabyte disk input/output operations are commonly a major bottleneck in large application codes on distributed-memory parallel supercomputers. Our first attempt to remove this bottleneck produced `DONIO` [2], a library of routines to provide fast parallel file I/O capabilities on Intel iPSC/860 and Intel Paragon supercomputers. `DONIO` caches the *entire* disk file across the aggregate memory of the multiprocessor in shared memory emulated by `DOLIB` (Distributed Object Library). This approach imposed a high memory overhead, and the use of 32-bit integer indexing restricted access to files of at most 2Gigabytes. The new `EDONIO` library reduces memory overhead and provides fast I/O on files of arbitrary size. `EDONIO` is implemented independently of the Distributed Object Library `DOLIB` [1] but uses similar `IPX` remote procedure calls to implement a large disk cache in the aggregate memory of the multiprocessor.

In contrast to `DONIO` where the *entire* file is cache in memory and actual disk I/O was done only in three routines (`do_open`, `do_flush` and `do_close`), `EDONIO` caches only a portion of the disk file. At runtime, as the limited disk cache is filled, data are immediately written back to the disk in contiguous large blocks of optimal size (default is 64Kbytes to match the RAID striping parameter) for high I/O throughput. Similarly, data not found in the disk cache is dynamically read in large blocks.

The amount of memory dedicated to `EDONIO` is controlled by the user. A larger disk cache usually results in better performance; especially if sufficient memory is available to cache the entire file into memory. In this case `EDONIO` reverts back to the behavior of `DONIO`.

## 2. Extended Distributed Object Network I/O Library

`EDONIO`, like `DONIO`, is designed to speed up the I/O for distributed-memory parallel applications where all processors open a *common* multi-megabyte shared file for simultaneous access. To access a shared file, each processor positions 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 iPSC/860 and Paragon attempt to support simultaneous access through a special file system (CFS for the iPSC/860, 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. It is common for file I/O to be one of the most costly operations in a parallel application. On the Intel Paragon machines, the default `M_UNIX` mode corresponds to standard UNIX file sharing semantics that enforce atomic updates by serializing all requests. The new `M_ASYNC` file I/O mode allows multiple simultaneous read/write requests with no restrictions and dramatically reduces the cost of I/O operations over the previous `M_UNIX` mode. `EDONIO` is designed to fully exploit the parallel `M_ASYNC` I/O mode by allowing all processors to perform non-overlapping I/O requests. Moreover, `EDONIO` uses the aggregate memory of the multiprocessor to implement a very large high-speed disk cache.

`EDONIO` is compatible with `DONIO` and 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 3 describes the use of these `EDONIO` primitives in more detail. Changing the names of the I/O subroutines called in an application program from the `NX` version to the `EDONIO` version (leaving the parameters untouched) and then linking in the `EDONIO` library is generally all that is required to use the package. *An important note:* `EDONIO` operates only on UNIX binary files, which may be incompatible with Fortran unformatted fixed-size record files.

Many large-scale applications involving the simulation of time-evolving events are designed to output a “snapshot” or “checkpoint” of the current state of the simulation at regular intervals. A lengthy simulation may output tens (or even hundreds) of

Gigabytes of data for later analysis. The original `DONIO` was incapable of handling files larger than 2Gigabytes. `EDONIO` overcomes this restriction, thereby providing rapid I/O capabilities on files of practically unlimited size (up to 16Terabytes).

### **3. User Interface**

The following pages provide details on the syntax and behavior of each of the `EDONIO` primitives. These pages can be considered the manual for `EDONIO`.

## `do_check()`

`do_check()` checks the message queues for EDONIO or IPX requests from other processors, servicing any that are found.

### Synopsis

```
int docheck( )
```

```
subroutine docheck( )
```

### Discussion

`do_check()` checks the calling processor's message queues for IPX requests from other processors. If none are found, `do_check()` returns immediately. Any queued requests are serviced before `do_check()` terminates. `do_check()` is provided to allow the user to avoid deadlock or slow servicing (starvation) of I/O requests if a non-interrupt (polling) version of IPX is used. All EDONIO calls automatically perform a `do_check()` operation. However, `do_check()` should be called periodically by processors that are not involved in file I/O operations for long periods of time.

## 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 all buffered 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 caching the file data associated with the file descriptor `fd`. For write-only and read-write files, `do_close()` first calls `do_flush()` to write out any cached data to the disk file before resources are deallocated. (If none of the cached pages are dirty, or if the file is read-only, 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 changes made to cached blocks that have not yet been flushed will be lost upon program termination!* All processors must participate in the `do_close()` call. An implicit global synchronization is performed.

## do\_csize()

`do_csize()` sets the sizes of the EDONIO read-only data cache and disk cache. An implicit global synchronization is performed.

### Synopsis

```
int do_csize( int data_size, int disk_size )  
  
subroutine docsiz( datasize, disksiz )  
integer datasize, disksiz
```

### Input parameters

- `data_size` - `data_size` is the maximum amount of memory in KBytes to be allocated to the read-only data cache. A value of 0 is valid, and can be used to disable the read-only cache if no user files are opened with permission flag `O_RDONLY`.
- `disk_size` - `disk_size` is the maximum amount of memory in KBytes to be allocated to the disk cache. A value of 0 results in an error.

### Discussion

`do_csize()` determines the *maximum* memory usage allowed by EDONIO's read-only data cache and disk cache. Actual allocation of memory for the caches is done only as needed. *Tip:* The user might call `vm_statistics()` at runtime or use `vm_stat` on the service nodes to determine the amount of free memory (or free pages) available. To avoid excessive paging, parameters for `do_csize()` should not exceed the amount of free memory.

All processors must participate in the `do_csize()`. An implicit global synchronization is performed.

## do\_flush()

`do_flush()` forces EDONIO to write any “dirty” or “modified” blocks associated with the specified file to the disk. After `do_flush()`, the disk file and cached blocks are guaranteed to be consistent. 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 any dirty blocks corresponding to the specified file to disk. If no changes have been made to the cached file since the last call to `do_flush()`, no disk I/O will take place. `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. EDONIO 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.

`do_flush()` may also enhance performance of write operations. If a cache miss causes EDONIO to flush a dirty cache block, only that block is written to disk. Better I/O performance may be obtained by writing many blocks concurrently with `do_flush()`.

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

## `do_lsize()`, `do_esize()`

`do_lsize()` estimates the size of the write-only or read-write output file associated with file descriptor `fd`. In C, `do_lsize()` returns `nbytes` on success. An implicit global synchronization is performed.

### Synopsis

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

```
esize_t do_esize( int fd, esize_t nbytes )
```

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

```
integer fd, nbytes
```

```
subroutine doesize( fd, lnbytes )
```

```
integer fd, lnbytes(2)
```

### Input parameters

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

### Discussion

`do_lsize()` tries to increase I/O throughput by attempting to preallocate the requested disk blocks before starting write operations. Unlike `DONIO` it is no longer mandatory to call `do_lsize()`. Overestimation of the file size may cause overallocation and suboptimal performance, but the actual file generated on disk will be of correct (minimal) size. 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\_eseek()

*do\_lseek()* (*do\_eseek()*) sets the (local) seek pointer of the open file associated with the file descriptor and returns the new seek position.

### Synopsis

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

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

subroutine doseek( fd, loffset, whence, lpos )
integer fd, whence
integer loffset(2), lpos(2)
```

### Input parameters

- fd** – **fd** is the file descriptor obtained from `do_open()`.
- offset** – **offset** is the offset in bytes. Note that EDONIO supports extended files larger than 2Gigabytes. For these extended files, the offset and returned value must be an extended integer (`esize_t`) in C, or an integer array of length 2 in FORTRAN.
- whence** – **whence** determines the computation with offset. **whence** is one of `SEEK_SET=0`, `SEEK_CUR=1` or `SEEK_END=2`.

## Discussion

`do_lseek()` (`do_eseek()`) 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`. *IMPORTANT NOTE: Calling `do_lseek()` using `whence=SEEK_END` is guaranteed correct only in two cases: the file must have been opened with `O_RDONLY`, or a call to `do_flush()` must immediately precede the `do_lseek()` call.* The reason is that the current file size has no meaning until all buffered writes have been flushed.

`dolseek( fd, 0, SEEK_END)` (after `do_flush()`, as described above) returns the size (in bytes) of the opened file associated with `fd`.

## `do_nio()`

`do_nio()` initializes the EDONIO 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

All nodes must call `do_nio()` to initialize the EDONIO network I/O library. `do_nio()` sets up internal data structures and initializes the IPX subsystem. Calling `do_nio()` is required before any other calls to EDONIO 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.
- `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.

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 on file  
c ---  
      mode = 8*8*6 + 8*6 + 6  
c ---  
c --- UNIX flags  
c --- O_RDONLY = 0, O_WRONLY = 1, O_RDWR = 2  
c ---  
      rflags = 0  
      wflags = 1  
      rwflags = 2  
c ---  
c --- be sure path is null terminated  
c ---  
      path = '/pfs/infile' // char(0)  
  
c ---  
c --- open the file for read-write access  
c ---  
      fd = doopen( path, rwflags, mode )
```

## do\_preload()

*do\_preload()* fills any empty slots in the cache with blocks from the disk file, starting with the first block referenced by the minimum value of all the local seek pointers.

*An implicit global synchronization is performed.*

### Synopsis

```
void do_preload( int fd )  
  
subroutine dopreload( fd )  
integer fd
```

### Input parameters

**fd** – **fd** is the EDONIO file descriptor for the file opened with `do_open()`.

### Discussion

`do_preload()` fills any empty slots in the disk cache with data from the disk. Preloading the cache is desirable when file access patterns may cause disk I/O to be inefficient. For example, if a number of processors attempt to read common data from the same processor, then there may be significant idle time while they all wait for the data to be brought in from disk. Preloading the cache ensures that the initial disk I/O is fully parallel and subsequent read accesses can proceed at full speed from the disk cache. Preloading starts from the point of the minimum seek location among all processors. The user can perform a `do_lseek()` (`do_esseek()`) immediately prior to the `do_preload()` call to ensure that the data in the cache are relevant to subsequent operations. By default, preloading starts from the beginning of file.

Note that preloading will not displace data already in the disk cache. In particular, if the cache is already full, then `do_preload()` has no effect. However, the user can force the creation of empty slots either by calling `do_csize()` to increase the memory allocated for the cache, or alternatively, the user can force a partial

purge of the cache by using two consecutive `do_csize()` calls to contract and then reset the disk cache size.

All processors must participate in the `do_preload()` call. An implicit global synchronization is performed.

## 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)`).

The calling process waits (blocks) until the request is completed. *Important:* Note that reading past the end of file causes an *error* instead of partially filling the buffer. 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.

Note that the execution times for the `do_read()` may vary substantially, depending on the access pattern and effectiveness of the disk cache.

## 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)`).

The calling process waits (blocks) until the request is completed. Using `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.

Note that the execution times for `do_write()` may vary significantly, depending on the access pattern and effectiveness of the disk cache.

## 4. Implementation Details

EDONIO provides a large high-speed disk cache in the aggregate memory of the Intel multiprocessor. The most important difference between EDONIO and DONIO is that the *entire* disk file is no longer kept in memory as in DONIO. Instead, EDONIO acts more as a true disk cache, reading and writing blocks of the file as needed. Hence EDONIO no longer requires the user to call `do_lsize()` before `do_write()`. `do_lsize()` (`do_esize()`) is now merely a hint to the operating system concerning the eventual file size. EDONIO automatically keeps track of the highest address actually used. If the user overestimates the file size in `do_lsize()` (`do_esize()`), then the correct (exact) size file will still be written to disk.

The conceptual view of a disk file in EDONIO is a sequence of blocks, each containing a fixed number (default 8 pages) of fixed size (default 8KBytes) pages.<sup>1</sup> Responsibility for actual disk I/O on the blocks is assigned to the processors in a wrap-mapped fashion. Thus, in an  $N$ -processor configuration, processor  $p$  is responsible for satisfying any I/O requests involving blocks  $p, p + N, p + 2N, \dots$  etc.

EDONIO supplies two separate caches: the *disk cache* and the *read-only data cache*. A processor's *disk cache* contains blocks of the disk file that have been most recently accessed. Note that blocks are only cached in the *disk cache* by the processor responsible for the given block, thus eliminating concerns for cache coherency. EDONIO also provides a *read-only data cache* for read-only files to reduce message traffic on repeated re-reads of the same data. Read-only files cannot be updated and is completely free from cache-coherency restrictions, therefore, the read-only data cache may hold any data that has been accessed, regardless of assignment (though the actual disk read is still performed by the assigned processor).

EDONIO uses the least recently used (LRU) strategy for cache management. That is, if the cache is full when a cache miss occurs, the least recently accessed block in the disk cache is deleted to make room for the incoming cache block. For the read-only data cache, merely freeing the memory is sufficient. However, for the disk cache, the chosen block is first checked to see if it is "dirty" (*i.e.*, has been altered). If so, it is written out to disk before it is deleted from the cache. This differs markedly from

---

<sup>1</sup>The xps35 Intel Paragon uses hardware pagesize of 8Kbytes, and RAID disk stripe size is configured to be 64Kbytes.

DONIO, where the cache was set large enough to contain the entire file, thus eliminating the need for disk I/O until the file was closed.

In EDONIO, all processors must participate concurrently in `do_open()`, `do_lsize()` (`do_esize()`), `do_flush()` and `do_close()`. The processors are synchronized when opening a shared file with `do_open()` so that EDONIO can set up common data structures. They are synchronized in `do_flush()` and in `do_close()` to ensure that there are no outstanding read or write requests.

EDONIO must deviate from the UNIX file system with respect to file permissions. The UNIX file systems allow a user to open an existing file with flag `O_WRONLY` (assuming the file mode allows write access) in a directory in which the user does not have read access. EDONIO cannot allow this, since it is impossible for EDONIO to act as a disk cache on a file without read permission. For simplicity, we assume that the user has read permission on any files that will be accessed with `do_open()`. Moreover, although EDONIO supports a write-only file mode (as a safety check to prevent read operations on the file), the *actual* file permissions must allow both reading and writing.

The original DONIO did not support an APPEND mode for file I/O. Instead, the user was advised to open separate files for each logically separate set of data, largely because of the inherent limitation on file size in DONIO. With EDONIO, the UNIX `O_APPEND` is still not directly supported but file size is no longer a concern, as we now fully support files of practically unlimited size (up to 16Terabytes). The user can append to a file by first seeking to end of file (see description on `do_lseek()` and `do_flush()`) before writing.

With EDONIO, the execution times for `do_read()` and `do_write()` may vary significantly depending on the ratio of cache hits/misses. The user can reduce these times in several ways. The size of the cache can be increased (see `do_csize()`) to improve the probability of cache hits, or preloading of the cache (see `do_preload()`) can also improve I/O performance.

Consider the sequence of events initiated by a `do_read()` request. First, the disk blocks involved are identified. If the disk block is assigned to the same calling processor, the local disk cache is searched. A cache miss causes EDONIO to load these blocks into the local disk cache, displacing other blocks if necessary. For any blocks assigned to

other processors, the IPX<sup>2</sup> system [3] is used to request the data from the processors that “own” those blocks. The read request is satisfied after the remote data are received. If the file was opened as a read-only file, the incoming data are also placed in the local read-only cache, to reduce message passing traffic should the same data be referenced in subsequent read operations. Note the read-only data cache holds only remote (non-local) data.

A `do_write()` operation is similar. Again, the disk blocks to be written are identified. Blocks assigned to the same processor are loaded into the cache if they are not already there. EDONIO uses the IPX ‘‘on’’ routine (a type of “remote procedure call”) to cause other processors to *update* blocks assigned to them. On the iPSC/860, IPX uses the NX `hrecv()` interrupt mechanism to preempt a processor to service IPX requests. However, on the Intel Paragon, `hrecv()` is not a true interrupt handler but spawns a separate thread that executes concurrently with the main computation. The extensive use of `masktrap()` for exclusive access to critical sections incurs a very high overhead on the Paragon. We have chosen to use a more efficient non-interrupt (polling) version of IPX for use on the Paragon. Because IPX requests are serviced only when the message queue is polled, and processors must supply data or update blocks at the request of other processors, the user must be careful to prevent deadlock or starvation. EDONIO provides the `do_check()` routine to examine the message queue for IPX requests. For example, code that uses a subset of the processors to handle all the disk I/O will fail unless the remaining processors periodically call `do_check()`, since IPX requests to these processors will not be serviced. See the manual page for `do_check()` for further discussion.

We have included a subprogram for preloading the disk-cache to enhance performance of the disk I/O. Preloading of the disk-cache is particularly desirable immediately after opening an existing file, where disk I/O during preloading proceeds in parallel. Preloading is not guaranteed to improve I/O performance since it depends on the access pattern and size of disk cache. See the manual page for `do_preload()` for further details.

---

<sup>2</sup>IPX is available by anonymous FTP from `msg.das.bnl.gov` under the directory `/pub/ipx`.

## 5. Experimental Results

In this section we present a rough comparison of disk performance by `EDONIO` 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, `cread()`'s and `cwrite()`'s. This example generates the element-to-vertex list for a three dimensional  $nex \times ney \times nez$  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 `xps35` Intel Paragon system at the Oak Ridge National Laboratory, and `EDONIO` is currently undergoing performance tuning, the performance numbers listed should be taken only as approximate and reflect only the current state of affairs (Feb 1995, OS version R1.2.5). Moreover, background disk activity by other concurrently running applications may also affect the timings. Three problems were used for testing: a small  $100 \times 100 \times 100$  (1,000,000 elements) problem, a medium  $200 \times 200 \times 200$  (8,000,000 elements), and a large  $300 \times 300 \times 300$  (27,000,000 elements) problem.

Table 5.1 show the effect of varying the amount of memory allocated to the disk cache in `EDONIO` on 22 nodes on a  $200 \times 200 \times 200$  grid (file size is  $256 \times 10^6$  bytes). We see from Table 5.1 that optimal performance is obtained when the aggregate disk cache can hold the *entire* file. Table 5.2 shows preloading the disk cache can reduce I/O time in `read` for 16 nodes on  $121 \times 121 \times 91$  grid (file size is 42,634,592 bytes). Runtimes are obtained from `dclock()`.

Tables 5.3–5.5 list the runtimes (in seconds) for the three problems. All runs have `EDONIO` configured to use 512Kbytes for read-only data cache, 4096Kbytes for disk cache and with cache preloading. Note that with the default 4096Kbytes allocated for the disk cache, 8, 62 and 206 processors are needed to hold the small, medium and large problems (respectively) in memory. The label `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 `read` and `write` times in `EDONIO` decrease with the addition of more processors. As more processors are used, fewer messages *per* processor

Table 5.1: Effect on disk cache size on EDONIO, all times in seconds.

Cache (KBytes)	wopen	write	wclose	ropen	preload	read	rclose
1024	3.0	20.6	0.7	1.6	0.7	55.6	0.2
2048	1.7	21.2	1.5	1.2	1.3	52.7	0.3
4096	1.6	17.3	3.2	2.2	2.6	46.2	0.3
8192	1.5	13.4	6.2	1.2	5.5	29.4	0.3
12288	1.6	9.6	6.6	2.2	10.3	20.3	1.5

Table 5.2: Effect of `do_preload()` on EDONIO, all times in seconds.

	wopen	write	wclose	ropen	preload	read	rclose
With preload	2.2	3.1	2.4	1.3	1.3	4.7	0.2
No preload	1.2	3.1	1.8	0.8	0.0	25.3	0.2
NX	1.4	38.5	0.2	0.7	0.0	25.4	0.2

are generated. Moreover, more total aggregate memory (4Mbytes per processor) is available for the disk cache. `wclose` and `preload` involve physical disk activity to write out or read in data into the aggregate disk cache; hence as the disk cache size increases with more processors, more data are transferred and more time for disk I/O may be required.

We see that with a large enough disk cache, EDONIO may offer nearly a ten-fold improvement over native NX routines. However, if the disk cache is too small to be effective, performance of EDONIO may be similar to native NX. EDONIO fully exploits the new `M_ASYNC` mode in achieving over 20Megabytes per second overall disk throughput to the `/pfs`. By comparison, DONIO with the default `M_UNIX` mode obtained only about 5Megabytes per second disk throughput.

## 6. Summary

We have described EDONIO, a fast file I/O emulation library for the Intel iPSC and Paragon distributed memory multiprocessors. EDONIO provides an easy to use interface, and with minimal change to the source of an iPSC/860 or Paragon parallel program may improve file I/O by a ten-fold speedup. Similar to the shared-memory library DOLIB, EDONIO uses the IPX message system to provide a very large high-speed disk

Table 5.3: Runtimes (in seconds) of EDONIO (NX) routines on  $100 \times 100 \times 100$  grid, file size is  $32 \times 10^6$  bytes.

processor	wopen	write	wclose	ropen	preload	read	rclose
1	1.9 (1.0)	29.1 (293.7)	0.7 (0.1)	0.7 (0.4)	2.5	44.9 (213.7)	0.1 (0.1)
2	1.5 (1.0)	20.9 (187.8)	0.6 (0.1)	0.6 (0.5)	2.8	39.5 (83.8)	0.1 (0.1)
4	2.0 (0.9)	13.2 (84.2)	0.7 (0.1)	0.6 (0.6)	2.8	22.6 (48.9)	0.1 (0.1)
8	1.4 (0.9)	6.4 (50.5)	1.2 (0.1)	0.7 (0.6)	1.9	8.2 (34.2)	0.2 (0.1)
16	1.2 (1.5)	3.5 (26.8)	1.1 (0.2)	1.0 (0.7)	1.8	4.3 (17.0)	0.3 (0.2)
32	2.0 (1.6)	2.0 (20.5)	1.8 (0.4)	1.3 (1.4)	1.0	2.3 (10.0)	0.4 (0.4)
64	3.1 (2.8)	1.3 (22.6)	2.4 (0.7)	2.5 (3.0)	0.9	1.3 (9.5)	0.8 (0.9)

Table 5.4: Runtimes (in seconds) of EDONIO (NX) routines on  $200 \times 200 \times 200$  grid, file size is  $256 \times 10^6$  bytes.

processor	wopen	write	wclose	ropen	preload	read	rclose
16	3.1(1.3)	31.3(141.2)	1.9(0.2)	1.4(0.8)	2.2	84.5(89.5)	0.3(0.2)
32	3.1(2.1)	15.5(122.4)	3.2(0.4)	1.5(1.3)	3.6	30.1(49.3)	0.4(0.4)
64	3.0(3.5)	5.3(118.6)	7.6(0.7)	2.5(2.1)	7.9	7.2(48.0)	0.8(0.7)
128	4.7(4.7)	3.0(89.2)	10.7(1.5)	4.3(3.7)	7.7	4.0(47.5)	1.6(1.4)

Table 5.5: Runtimes (in seconds) of EDONIO (NX) routines on  $300 \times 300 \times 300$  grid, file size is  $864 \times 10^6$  bytes.

processor	wopen	write	wclose	ropen	preload	read	rclose
32	2.1(1.5)	45.9(262.0)	5.2(0.4)	2.6(2.3)	3.4	119.5(111.4)	0.4(0.3)
64	2.9(2.8)	24.1(218.1)	7.3(0.7)	2.8(2.2)	6.5	56.7(108.8)	0.8(0.7)
128	4.9(4.5)	14.1(360.3)	23.1(1.5)	4.6(4.8)	15.8	21.4(105.2)	1.5(1.5)

cache in the aggregate memory of the multiprocessor. Disk I/O operations are in large blocks to fully exploit the new `M_ASYNC` I/O mode. `EDONIO` is more memory efficient than `DONIO` and can access files of practically unlimited size.

## **7. Obtaining the Software**

To obtain the source code for `EDONIO` the reader should send email to the authors: `e6d@ornl.gov` or `rominech@ornl.gov`.

## **Acknowledgments**

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

## 8. Appendix

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

```

        program  example
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 M_MODE M_ASYNC

#define IOINIT(myid,nproc)
#define LSEEK lseek
#define ROPEN(fd, filename) call gopen(fd,filename,M_MODE)
#define WOPEN(fd, filename) call gopen(fd,filename,M_MODE)
#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() call 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) ierr = doread(fd, ibuffer, nbytes )
#define CWRITE(fd, ibuffer, nbytes) ierr = dowrite( fd, ibuffer, nbytes )
#define CCLOSE( fd ) call doclose(fd)

#define GSYNC() call dogsync()
#endif

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

integer data_size,disk_size
integer ipreload

double precision tstart,tend
character*80 filename
integer i, ix,iy,iz, nnx,nny,nnz, nex,ney,nez
integer jx,jy,jz
integer mbuf(nvertex,maxnez)
integer mbuf2(nvertex,maxnez)
integer nbytes, myid,nproc,ihost
real*8 totalbytes
integer mi,miold,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()

#if RX || I860
call open0(nproc, myid, ihost )
#endif
IOINIT( myid, nproc )
```

```
nex = 0
ney = 0
nez = 0

data_size = 0
disk_size = 0

ipreload = 0

if (myid .eq. 0) then
  write(outdev,*) 'enter nex,ney,nez '
  read(indev,*) nex,ney,nez
  write(outdev,*) 'nproc, nex,ney,nez ', nproc,nex,ney,nez

  write(outdev,*) 'enter data_size, disk_size (in Kbytes)'
  read(indev,*) data_size,disk_size
  write(outdev,*) 'data_size,disk_size',data_size,disk_size

  write(outdev,*) 'enter use of preload '
  read(indev,*) ipreload
  write(outdev,*) 'ipreload ',ipreload

endif

call gisum( data_size, 1, iwork )
call gisum( disk_size, 1, iwork)

call docsize( data_size, disk_size )

call gisum( ipreload, 1, iwork )

call gisum(nex,1,iwork)
call gisum(ney,1,iwork)
call gisum(nez,1,iwork)

  nnx = nex + 1
  nny = ney + 1
  nnz = nez + 1

totalbytes = dble(nex*ney*nez)*dble(nvertex*sizeint)

GSYNC()
tstart = dclock()
#ifdef USE_NX
#if RX || I860
filename = '/cfs/nxex.bin'
```

```
#else
    filename = '/pfs/nxex.bin'
#endif

#else /* USE_NX */
c---
c---    IMPORTANT NOTE: string MUST be null terminated
c---
#if RX || I860
    filename = '/cfs/ex.bin' // char(0)
#else
    filename = '/pfs/ex.bin' // char(0)
#endif

#endif /* USE_NX */

    WOPEN( fd, filename )

    GSYNC()
    tend = dclock()
    if (myid .eq. 0) then
        write(outdev,*) ' open takes ', tend-tstart,' sec'
        write(outdev,*) ' total file size is ',
&                int(totalbytes/1024.0/1024.0),' Megbytes'
    endif
c
    nbytes = nvertex*sizeint
    GSYNC()
    tstart = dclock()

    miold = -1
    do iy=1,ney
    do ix=1,nex

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

        if (ismine) then
            do iz=1,nez
                do i=1,nvertex
                    jx = ix+dx(i)
                    jy = iy+dy(i)
                    jz = iz+dz(i)
                    mbuf(i,iz)=ijk2ni(jx,jy,jz,nnx,nnz)
                enddo
            enddo

            mi = ijk2mi( ix,iy,1,  nex,ney,nez)
            if (miold.eq.-1) then
                offset = (mi-1)*nvertex*sizeint
                ierr = LSEEK( fd, offset, SEEK_SET )
            endif
        endif
    enddo
enddo
```

```
        else
            offset = (mi-miold)*nvertex*sizeint - nbytes
            ierr = LSEEK( fd, offset, SEEK_CUR )
        endif
        miold = mi
        nbytes = nez*nvertex*sizeint
        CWRITE( fd, mbuf(1,1), nbytes )
    endif
enddo
enddo

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

GSYNC()
tstart = dclock()
CCLOSE( fd )
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 ---

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

if (ipreload.ne.0) then
    GSYNC()
    tstart = dclock()
    call dopreload( fd )
    GSYNC()
    tend = dclock()
    if (myid.eq.0) then
        write(outdev,*) 'preload takes ',tend-tstart,' sec'
    endif
endif
endif
```

```
nbytes = nvertex*sizeint
GSYNC()
tstart = dclock()

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

  if (ismine) then
    do iz=1,nez
      do i=1,nvertex
        jx = ix+dx(i)
        jy = iy+dy(i)
        jz = iz+dz(i)
        mbuf2(i,iz)=ijk2ni(jx,jy,jz,nnx,nnz)
      enddo
    enddo
  endif

  if (ismine) then
    mi = ijk2mi( ix,iy, 1,    nex,ney,nez)
    if (miold.eq.-1) then
      offset = (mi-1)*nvertex*sizeint
      ierr = LSEEK( fd, offset, SEEK_SET )
    else
      offset = (mi-miold)*nvertex*sizeint - nbytes
      ierr = LSEEK( fd, offset, SEEK_CUR )
    endif
    miold = mi
    nbytes = nez*nvertex*sizeint
    CREAD( fd, mbuf(1,1), nbytes )
  endif

c ---
c --- double check results
c ---

  if (ismine) then
    do iz=1,nez
      do i=1,nvertex
        if (mbuf2(i,iz).ne.mbuf(i,iz)) then
          write(*,9900) i,iz,mbuf2(i,iz),mbuf(i,iz)
9900          format('i,iz,mbuf2(i,iz),mbuf(i,iz)',4(1x,i7))

          stop '** ERROR ** '
        endif
      enddo
    enddo
  endif
endif
```

```
enddo  
enddo
```

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

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

```
stop  
end
```

## 9. 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] ———, *DONIO: Distributed Object Network I/O Library*, Tech. Report ORNL/TM-12743, Oak Ridge National Laboratory, 1994.
- [3] B. MARR, R. PEIERLS, AND J. PASCIAK, *IPX – Preemptive remote procedure execution for concurrent applications*, Tech. Report, Brookhaven National Laboratory, 1994.

**INTERNAL DISTRIBUTION**

- |                      |  |
|----------------------|--|
| 1. B. R. Appleton    | 28. B. A. Riley                          |
| 2. B. A. Carreras    | 29-33. C. H. Romine                      |
| 3-7. E. F. D'Azevedo | 34. W. A. Shelton                        |
| 8. T. S. Darland     | 35-39. R. F. Sinovec                     |
| 9. J. J. Dongarra    | 40. G. M. Stocks                         |
| 10. J. B. Drake      | 41. M. R. Strayer                        |
| 11. T. H. Dunigan    | 42. D. W. Walker                         |
| 12. W. R. Emanuel    | 43. P. H. Worley                         |
| 13. G. A. Geist      | 44. T. Zacharia                          |
| 14. K. L. Kliewer    | 45. Central Research Library             |
| 15-19. M. R. Leuze   | 46. ORNL Patent Office                   |
| 20. E. G. Ng         | 47. K-25 Applied Technology Li-<br>brary |
| 21. C. E. Oliver     | 48. Y-12 Technical Library               |
| 22. B. W. Peyton     | 49. Laboratory Records - RC              |
| 23-27. S. A. Raby    | 50-51. Laboratory Records Dept.          |

**EXTERNAL DISTRIBUTION**

52. Loyce M. Adams, Applied Mathematics, FS-20, University of Washington, Seattle, WA 98195
53. Christopher R. Anderson, Department of Mathematics, University of California, Los Angeles, CA 90024
54. Todd Arbogast, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston, TX 77251
55. Donald M. Austin, 6196 EECS Building, University of Minnesota, 200 Union Street, S.E., Minneapolis, MN 55455
56. Robert G. Babb, Oregon Graduate Center, CSE Department, 19600 N.W. Walker Road, Beaverton, OR 97006
57. David H. Bailey, NASA Ames, Mail Stop 258-5, NASA Ames Research Center, Moffet Field, CA 94035
58. Jesse L. Barlow, Department of Computer Science and Engineering, 220 Pond Laboratory, The Pennsylvania State University, University Park, PA 16802-6106
59. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratory, Albuquerque, NM 87185
60. Adam Beguelin, Carnegie Mellon University, School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213-3890
61. Robert E. Benner, Parallel Processing Division 1413, Sandia National Laboratories, P. O. Box 5800, Albuquerque, NM 87185

62. Marsha J. Berger, Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012
63. Philippe Berger, Institut National Polytechnique, ENSEEIHT, 2 rue Charles Camichel-F, 31071 Toulouse Cedex, France
64. Ake Bjorck, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden
65. John H. Bolstad, L-16, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
66. Roger W. Brockett, Harvard University, Pierce Hall, 29 Oxford Street Cambridge, MA 02138
67. James C. Browne, Department of Computer Sciences, University of Texas, Austin, TX 78712
68. Bill L. Buzbee, Scientific Computing Division, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307
69. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109
70. Thomas A. Callcott, Director, The Science Alliance Program, 53 Turner House, University of Tennessee, Knoxville, TN 37996
71. Tony Chan, Department of Mathematics, University of California, Los Angeles, 405 Hilgard Avenue, Los Angeles, CA 90024
72. Jagdish Chandra, Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709
73. Siddhartha Chatterjee, Dept. of Computer Science, CB 3175, Sitterson Hall, The University of North Carolina, Chapel Hill, NC 27599-3175
74. Melvyn Ciment, National Science Foundation, 1800 G Street N.W., Washington, DC 20550
75. Tom Coleman, Department of Computer Science, Cornell University, Ithaca, NY 14853
76. Alva Couch, Department of Computer Science, Tufts University, Medford, MA 02155
77. Paul Concus, Mathematics and Computing, Lawrence Berkeley Laboratory, Berkeley, CA 94720
78. Tom Crockett, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665-5225
79. Jane K. Cullum, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598
80. George Cybenko, Center for Supercomputing Research and Development, University of Illinois, 104 South Wright Street, Urbana, IL 61801-2932
81. Helen Davis, Computer Science Department, Stanford University, Stanford, CA 94305
82. Michel Dayde, Institut National Polytechnique, ENSEEIHT, 2 rue Charles Camichel-F, 31071 Toulouse Cedex, France

83. Craig Douglas, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598-0218
84. Iain S. Duff, Atlas Centre, Rutherford Appleton Laboratory, Chilton, Oxon OX11 0QX, England
85. Victor Eijkhout, University of Tennessee, 107 Ayres Hall, Department of Computer Science, Knoxville, TN 37996-1301
86. Stanley Eisenstat, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
87. Howard C. Elman, Computer Science Department, University of Maryland, College Park, MD 20742
88. Richard E. Ewing, Director, Institute for Scientific Computations, Texas A&M University, College Station, TX 77843-3404
89. Edward Felten, Department of Computer Science, University of Washington, Seattle, WA 98195
90. Charles Fineman, Ames Research Center, Mail Stop 269/3, Moffet Field, CA 94035
91. David Fisher, Department of Mathematics, Harvey Mudd College, Claremont, CA 91711
92. Jon Flower, Parasoft Corporation, 2500 E. Foothill Boulevard, Suite 205, Pasadena, CA 91107
93. Geoffrey C. Fox, NPAC, 111 College Place, Syracuse University, Syracuse, NY 13244-4100
94. Chris Fraley, Statistical Sciences, Inc., 1700 Westlake Avenue N, Suite 500, Seattle, WA 98119
95. Joan M. Francioni, Computer Science Department, University of Southwestern Louisiana, Lafayette, LA 70504
96. Paul O. Frederickson, ACL, MS B287, Los Alamos National Laboratory, Los Alamos, NM 87545
97. Offir Frieder, George Mason University, Science and Technology Building, Computer Science Department, 4400 University Drive, Fairfax, Va 22030-4444
98. Robert E. Funderlic, Department of Computer Science, North Carolina State University, Raleigh, NC 27650
99. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47401
100. C. William Gear, NEC Research Institute, 4 Independence Way, Princeton, NJ 08540
101. W. Morven Gentleman, Division of Electrical Engineering, National Research Council, Building M-50, Room 344, Montreal Road, Ottawa, Ontario, Canada K1A 0R8
102. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

103. James Glimm, SUNY-Stony Brook, Department of Applied Mathematics and Statistics, Stony Brook, NY 11794
104. Gene Golub, Computer Science Department, Stanford University, Stanford, CA 94305
105. Joseph F. Grcar, Division 8245, Sandia National Laboratories, Livermore, CA 94551-0969
106. William D. Gropp, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
107. Eric Grosse, AT&T Bell Labs 2T-504, Murray Hill, NJ 07974
108. Sanjay Gupta, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665-5225
109. John L. Gustafson, Ames Laboratory, 236 Wilhelm Hall, Iowa State University, Ames, IA 50011-3020
110. Christian Halloy, Assistant Director of JICS, 104 South College, Joint Institute for Computational Science, University of Tennessee, Knoxville, TN 37996-1301
111. Sven J. Hammarling, The Numerical Algorithms Group, Ltd., Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, United Kingdom
112. Robert M. Haralick, Department of Electrical Engineering, Director, Intelligent Systems Lab, University of Washington, 402 Electrical Engineering Building, FT-10, Seattle, WA 98195
113. Ann H. Hayes, Computing and Communications Division, Los Alamos National Laboratory, Los Alamos, NM 87545
114. Michael T. Heath, National Center for Supercomputing Applications, 4157 Beckman Institute University of Illinois, 405 North Mathews Avenue, Urbana, IL 61801-2300
115. Gerald W. Hedstrom, L-71, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
116. Don E. Heller, Ames Laboratory, 327 Wilhelm, Ames, IA 50011
117. John L. Hennessy, CIS 208, Stanford University, Stanford, CA 94305
118. N. J. Higham, Department of Mathematics, University of Manchester, Gtr Manchester, M13 9PL, England
119. Dan Hitchcock, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
120. Charles J. Holland, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332
121. Fred Howes, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, Department of Energy, Washington, DC 20585
122. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
123. Jenq-Neng Hwang, Department of Electrical Engineering, FT-10, University of Washington, Seattle, WA 98195

124. Ilse Ipsen, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
125. Leah H. Jamieson, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907
126. Gary Johnson, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
127. Lennart Johnsson, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214
128. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
129. Bo Kagstrom, Institute of Information Processing, University of Umea, 5-901 87 Umea, Sweden
130. Malvyn Kalos, Cornell Theory Center, Engineering and Theory Center Building, Cornell University, Ithaca, NY 14853-3901
131. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Bldg. 221, Argonne, IL 60439
132. Alan H. Karp, HP Labs 3U-7, Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, CA 94304
133. Linda Kaufman, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
134. Robert J. Kee, Applied Mathematics Division 8245, Sandia National Laboratories, Livermore, CA 94551-0969
135. Kenneth Kennedy, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77001
136. Tom Kitchens, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
137. Clyde P. Kruskal, Department of Computer Science, University of Maryland, College Park, MD 20742
138. Edward Kushner, Intel Corporation, 15201 NW Greenbrier Parkway, Beaverton, OR 97006
139. Michael Langston, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301
140. Richard Lau, Office of Naval Research, Code 111MA 800 Quincy Street, Boston Tower 1, Arlington, VA 22217-5000
141. Robert L. Launer, Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709
142. Tom Leighton, Lab for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139
143. Robert Leland, Sandia National Laboratories, 1424, P. O. Box 5800, Albuquerque, NM 87185-5800

144. Randall J. LeVeque, Applied Mathematics, FS-20, University of Washington, Seattle, WA 98195
145. John G. Lewis, Boeing Computer Services, P. O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346
146. Heather M. Liddell, Center for Parallel Computing, Department of Computer Science and Statistics, Queen Mary College, University of London, Mile End Road, London E1 4NS, England
147. Brent Lindquist, SUNY-Stony Brook, Department of Applied Mathematics and Statistics, Stony Brook, NY 11794
148. Rik Littlefield, Pacific Northwest Laboratory, MS K1-87, P.O.Box 999, Richland, WA 99352
149. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, Downsview, Ontario, Canada M3J 1P3
150. Franklin Luk, Department of Computer Science, Amos Eaton Building — No. 131 Rensselaer Polytechnic Institute Troy, NY 12180-3590
151. Ewing Lusk, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, MCS 221 Argonne, IL 60439-4844
152. Allen D. Malony, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403
153. Thomas A. Manteuffel, Department of Mathematics, University of Colorado - Denver, Denver, CO 80202
154. Anita Mayo, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598
155. Oliver McBryan, University of Colorado at Boulder, Department of Computer Science, Campus Box 425, Boulder, CO 80309-0425
156. James McGraw, Lawrence Livermore National Laboratory, L-306, P. O. Box 808, Livermore, CA 94550
157. Piyush Mehrotra, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665
158. Paul C. Messina, Mail Code 158-79, California Institute of Technology, 1201 E. California Boulevard, Pasadena, CA 91125
159. Cleve B. Moler, MathWorks, 325 Linfield Place, Menlo Park, CA 94025
160. Jorge J. More, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
161. William A. Mulder, Koninklijke Shell Exploratie en Productie Laboratorium, Postbus 60, 2280 AB Rijswijk, The Netherlands
162. David Nelson, Director, Office of Scientific Computing, ER-7, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
163. V. E. Oberacker, Department of Physics, Vanderbilt University, Box 1807, Station B, Nashville, TN 37235

164. Dianne P. O'Leary, Computer Science Department, University of Maryland, College Park, MD 20742
165. Joseph Olinger, Computer Science Department, Stanford University, Stanford, CA 94305
166. James M. Ortega, Department of Computer Science, Thornton Hall, University of Virginia, Charlottesville, VA 22901
167. Steve Otto, Oregon Graduate Institute, Department of Computer Science and Engineering, 19600 NW von Neumann Drive, Beaverton, OR 97006-1999
168. Cherri Pancake, Department of Computer Science, Oregon State University, Corvallis, OR 97331-3202
169. Joseph E. Pasciak, Applied Mathematics, Brookhaven National Laboratory, Upton, NY 11973
170. Merrell Patrick, National Science Foundation, 1800 G Street N.W., Washington, DC 20550
171. David Payne, Intel Corporation, Supercomputer Systems Division, 15201 NW Greenbrier Parkway, Beaverton, OR 97006
172. Ronald F. Peierls, DAS — Bldg. 490-D, P.O. Box 5000, Brookhaven National Laboratory, Upton, NY 11973
173. Linda R. Petzold, Computer Science Department, University of Minnesota, 200 Union Street, S.E., Room 4-192, Minneapolis, MN 55455
174. Dan Pierce, Boeing Computer Services, P.O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346
175. Paul Pierce, Intel Corporation, Supercomputer Systems Division, 15201 NW Greenbrier Parkway, Beaverton, OR 97006
176. Robert J. Plemmons, Departments of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650
177. James C. T. Pool, Deputy Director, Caltech Concurrent Supercomputing Facility, California Institute of Technology, MS 158-79, Pasadena, CA 91125
178. Jesse Poore, Computer Science Department, University of Tennessee, Knoxville, TN 37996-1300
179. David A. Poplawski, Department of Computer Science, Michigan Technological University, Houghton, MI 49931
180. Roldan Pozo, University of Tennessee, 107 Ayres Hall, Department of Computer Science, Knoxville, TN 37996-1301
181. Padma Raghavan, University of Illinois, NCSA, 4151 Beckman Institute, 405 North Matthews Avenue, Urbana, IL 61801
182. Daniel A. Reed, Computer Science Department, University of Illinois, Urbana, IL 61801
183. John K. Reid, Numerical Analysis Group, Central Computing Department, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, England
184. John R. Rice, Computer Science Department, Purdue University, West Lafayette, IN 47907

185. Garry Rodrigue, Numerical Mathematics Group, Lawrence Livermore National Laboratory, Livermore, CA 94550
186. Donald J. Rose, Department of Computer Science, Duke University, Durham, NC 27706
187. Ahmed H. Sameh, Department of Computer Science, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455
188. Joel Saltz, Computer Science Department, A.V. Williams Building, University of Maryland, College Park, MD 20742
189. Jorge Sanz, IBM Almaden Research Center, Department K53/802, 650 Harry Road, San Jose, CA 95120
190. Robert B. Schnabel, Department of Computer Science, University of Colorado at Boulder, ECOT 7-7 Engineering Center, Campus Box 430, Boulder, CO 80309-0430
191. Robert Schreiber, RIACS, MS 230-5, NASA Ames Research Center, Moffet Field, CA 94035
192. Martin H. Schultz, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
193. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006
194. The Secretary, Department of Computer Science and Statistics, The University of Rhode Island, Kingston, RI 02881
195. Charles L. Seitz, Department of Computer Science, California Institute of Technology, Pasadena, CA 91125
196. Margaret L. Simmons, Computing and Communications Division, Los Alamos National Laboratory, Los Alamos, NM 87545
197. Horst D. Simon, NASA Ames Research Center, Mail Stop T045-1, Moffett Field, CA 94035
198. William C. Skamarock, 3973 Escuela Court, Boulder, CO 80301
199. Tony Skjellum, Dept of Computer Science, Mississippi State University, PO Drawer CS, Mississippi State, MS 39762-5623
200. Burton Smith, Tera Computer Company, 400 North 34th Street, Suite 300, Seattle, WA 98103
201. Marc Snir, IBM T.J. Watson Research Center, Department 420/36-241, P. O. Box 218, Yorktown Heights, NY 10598
202. Larry Snyder, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195
203. Danny C. Sorensen, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston, TX 77251
204. Rick Stevens, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
205. G. W. Stewart, Computer Science Department, University of Maryland, College Park, MD 20742

206. Paul N. Swarztrauber, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307
207. Julie Swishhelm, Sandia National Laboratories, 1421, Parallel Computational Sciences Department, Albuquerque, New Mexico 87185-5800
208. Wei Pai Tang, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2l 3G1
209. Bernard Tourancheau, LIP ENS-Lyon 69364, Lyon cedex 07, France
210. Joseph F. Traub, Department of Computer Science, Columbia University, New York, NY 10027
211. Lloyd N. Trefethen, Department of Computer Science, Cornell University, Ithaca, NY 14853
212. Robert van de Geijn, University of Texas, Department of Computer Sciences , TAI 2.124, Austin, TX 78712
213. Charles Van Loan, Department of Computer Science, Cornell University, Ithaca, NY 14853
214. Udaya B. Vemulapati, Department of Computer Science, University of Central Florida, Orlando, FL 32816-0362
215. Robert G. Voigt, National Science Foundation, Room 417, 1800 G Street N.W., Washington, DC 20550
216. Bi R. Vona, Center for Numerical Analysis, RLM 13.150, University of Texas at Austin, Austin, TX 78712
217. Henk A. van der Vorst, Professor Dept. of Mathematics, Universiteit Utrecht, P.O. Box 80010, 3508 TA, Utrecht, THE NETHERLANDS
218. Michael D. Vose, 107 Ayres Hall, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301
219. Phuong Vu, Cray Research, Inc., 19607 Franz Road, Houston, TX 77084
220. A. J. Wathen, School of Mathematics, University Walk, Bristol BS8 1TW, England
221. Robert P. Weaver, 1555 Rockmont Circle, Boulder, CO 80303
222. Mary F. Wheeler, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston, TX 77251
223. Andrew B. White, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545
224. John Zahorjan, Department of Computer Science and Engineering, Sieg Hall, FR-35, University of Washington, Seattle, WA 98195
225. Office of Assistant Manager for Energy Research and Development, U.S. Department of Energy, Oak Ridge Operations Office, P. O. Box 2001, Oak Ridge, TN 37831-8600
- 226-227. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831