Technical Report
RAL-TR-2001-020

CLRC

Parallel IO for High Performance Computing

C Greenough R F Fowler and R J Allan

22nd March 2001

COUNCIL FOR THE CENTRAL LABORATORY OF THE RESEARCH COUNCILS

Enquiries about copyright, reproduction and requests for additional copies of this report should be addressed to:

The Central Laboratory of the Research Councils
Library and Information Services
Rutherford Appleton Laboratory
Chilton
Didcot
Oxfordshire
OX11 0QX
Tel: 01235 445384 Fax: 01235 446403
E-mail library@rl.ac.uk

ISSN 1358-6254

Parallel IO for High Performance Computing

C.Greenough and R.F.Fowler

Computational Science and Engineering Department
CLRC Rutherford Appleton Laboratory,
Chilton, Didcot OX11 0QX, UK

R.J.Allan

Computational Science and Engineering Department
CLRC Daresbury Laboratory,
Warrington, WA4 4AD, UK

Email: c.greenough@rl.ac.uk or r.j.allan@dl.ac.uk

This report is available from http://www.cse.clrc.ac.uk/Activity/HPCI

February 21, 2001

Abstract

As the computational power of parallel systems increases there is a corresponding rise in the amount of data that they are required to process. In this paper we review some of the developments in commercial systems and research projects that are trying to address scientific and engineering needs for high performance IO systems.

Keywords: high performance computing, parallel IO

Contents

1 Introduction

High performance computing systems are often compared in terms of their floating point performance, communication speed and available main memory. However as applications become larger they also have to deal with more data and this can become just as much of a bottleneck as the actual computation. Therefore there is increasing interest in ways of speeding up Input/Output (IO) on such systems.

The performance of storage devices, such as hard disks, is increasing through developments such as faster rotational speed and higher data density on the magnetic media. However, these are not keeping pace with the demands of many important scientific and engineering applications. The amount of main memory available in typical high end computers is also increasing rapidly but not at a sufficient rate to satisfy many of the most demanding "grand challenge" style applications. Thus parallel IO is seen as a way of providing fast access to secondary storage for such problems.

Parallel IO is a very broad term that seems to cover the spectrum from hardware RAID [5] style disk arrays, where a single file can be striped across several disks, to software solutions such as MPI-IO, now part of MPI2 [4]. Clearly the underling hardware defines the maximum data transfer rates that can be achieved on a given machine. However, as with many performance characteristics of parallel computers, actually obtaining fast data transfers requires good software support at all levels. Typically the application program should have access to fast IO through a high level library built on efficient services provided by the operating system. Projects such as the Scalable IO Initiative [11] have been active in promoting such an integrated approach to support the data requirements of high performance computing. These efforts are yielding some results and most computer system vendors are now addressing this problem to some degree. Examples of this are the explicit support of parallel file systems from Intel, IBM and SGI and others, which are discussed later.

In the context of the new ideas in computational grids, such as Globus [13], high performance access to data sets across wide area networks is also important. Parallel IO concepts can also be useful for these applications, when large amounts of data need to be retrieved from multiple remote resources.

In this report we provide a brief review of the need for parallel IO and then go on to give some references to the products available in this area and related research projects. Several web sites already provide a great deal of information on parallel IO products and active research projects in this area. A recent review of parallel IO for scientific applications is given in [1]. The book "Parallel IO for High Performance Computing", by John May [2] gives a more complete overview of this area.

2 Overview of Parallel IO

2.1 IO Requirements for HPC

High performance computing applications often need to process vast amounts of data and of the course the wall clock time for the whole task must include these IO costs. Some estimates of the data IO requirements of typical applications (in 1994 [3]) are given in Table 1.

Application	Archive	Secondary	Bandwidth
Environmental Modelling	1 TB	1GB	-
Eulerian air quality	10 TB	100 GB	-
Earth system model	-	100 GB	100 MB/s
4D data assimilation	3TB	1GB	-
Ocean-atmosphere model	100TB	100GB	100MB/s
Solar activity model	500GB	-	200MB/s
Cosmology simulations	-	100GB	200MB/s

Table 1: Some order of magnitude estimates of IO requirements made in 1994. Secondary storage is usually fast disks on the parallel machine while archival storage may be a tape robot. Bandwidth figures are estimates of the required data transfer rates.

As well as reading and writing large amounts of data, some applications require significantly more working storage then is available on the individual nodes of the parallel machine. Some parallel machines do not support virtual memory while those that do may not perform well unless the solution algorithm is optimised to take account of the actual amount of real memory. Hence it is usually necessary to write explicit out-of-core [3] methods when solving problems larger than the physical memory size, and these of course will require high performance parallel IO.

The sort of IO rates mentioned in Table 1 cannot be obtained with currently available single disks, which have sustained rates of perhaps 40MB/s. It is also likely that desired transfer rates will have increased in the years since these estimates were made. Using the sum of GFLOPS of all machines in the Top500 list, computational power has increased by a factor of about 32 from 1994 to 2000, so required IO rates may now be an order of magnitude higher than the figures of Table 1. Therefore a parallel system needs to be able to use many storage devices concurrently to achieve the required data rates.

Another way of looking at IO requirements, described in [2], is in terms of the floating point operations (FLOPs) per byte of IO required. A suggested figure is about 500FLOP/byte, which would imply a machine capable of 100GFLOP/s would require 200MB/s IO, which is similar to the figures of Table 1. A machine running at 5000GFLOP/s (the highest figure in the current Top500 list) would require 10GB/s IO. Of course the FLOP/byte rate depends on the program and the IO requirements may not be uniform through out a run. Another significant observation in [2] is that, though both CPU speed and disk capacity have been increasing at roughly 60% per year recently, disk data transfer rates have only increased by 40% per year. Hence the IO bottleneck is expected to get worse.

2.2 Hardware for HPC IO

Use of multiple disks, such in RAID products can deliver high sustained transfer rates. In the simplest form, without any redundancy, an array of disks can be connected in parallel. A file can then be "striped" across this disk array so that the first block is written to disk1, the second

to disk2 and so on up to the N disk that are available. Further blocks are then written to disk1, disk2, etc. in the same order.

This method can achieve high performance as the data transfer rate increases linearly with the number of disk drives. Recently Sun has demonstrated a system with almost 1000 disk drives that can handle 3GB/s of data. These disks were not, however, in a single unit configuration.

One problem with connecting many disk drives in parallel is reliability. It would be unacceptable if the whole file system was lost every time one disk in an array failed. Hence disk arrays usually use RAID formats [5], where extra ("parity") data is written so that, in the case of a single disk failure, all the file system can be recovered. There is some performance cost in doing this, but this is not too great and it is important for the long run times required by many HPC applications. "Hot swappable" disks allow RAID arrays to have failing components replaced without the need to reboot the system.

Another problem with disk arrays is that of getting the data from the disks to the computational processors. The first bottleneck is the disk controller to which the devices are attached. For example a Wide Ultra2 SCSI interface permits 16 disks to be connected together with a maximum data transfer rate of 80MB/s. If individual disks are capable of 40MB/s then this can easily the total performance with more than 2 devices on the interface. Other interfaces exist such as SCSI Ultra3 (160MB/s), Fibre-Channel Arbitrated Loop (FC-AL, 100MB/s) and Serial Storage Architecture (SSA, 160 MB/s), but there is clearly a need to use more than one controller to allow scalable IO at the highest data rates.

The network to which the disk arrays are attached can also limit the IO performance. High speed Ethernet is limited to 10MB/s but other newer connection technologies such as SCI, Myrinet and HIPPI provide data rates in excess of 100MB/s. Even with high speed interconnect it is likely that IO devices will need to be distributed across the network to avoid bottlenecks.

The Storage Area Network (SAN) is a new technology to provide fast access to data storage. Several commercial manufacturers now provide SAN hardware and this appears to be a significant trend. Under SAN devices such as disk arrays and tapes are directly connected via a high speed connection technology such as Fibre Channel. This allows very fast communication up to distances of 10Km or more. Data can be transferred directly from the disk arrays to the host needing the data without having to pass through a "server" computer. Disk data can also be copied to tape directly across this network. More information about SAN technology is available at:
`//http://www.redbooks.ibm.com/abstracts/sg245470.html`.

To get high performance from hard disk arrays it is important that IO requests are made in large contiguous blocks due to the latency in reading disjoint areas of the disks. As the number of disks across which a file is striped increases the read size for maximum performance will also increase. In a typical case of a distributed application writing a single output file each processor will need to output its own section of the total file. Each individual output request may be too small to take full advantage of the parallelism of a large RAID array. Hence there is a lot of interest in software solutions, such as MPI-IO, which try to optimise the data transfer from multiple processors to a single output file.

The above comments mainly refer to disk storage directly attached to a single parallel machine. As heterogeneous networks of processors are connected together, in for example computational grids, the problem of providing efficient parallel IO becomes more complex. Solutions such as the High Performance Storage System (HPSS) [31] try to address the wider problem to data storage and archival. Using large RAID devices and tape robots for extensive backing store, HPSS provides faster data movement to and from the parallel machines. More details of HPSS are given in section 7.7.

3 Measurement and Benchmarking of Parallel IO Performance

In this section we describe some projects that are trying to measure and quantify the requirements of parallel IO for high performance scientific and engineering applications.

3.1 The Scalable IO Initiative

When the Scalable IO Initiative [11] (SIO) was established in 1994, one of the primary aims was to instrument and measure the IO performance of current applications. To this end they have developed IO performance monitoring tools, such as Pablo [12], to provide portable measurements across a range of parallel machines.

Three main types of IO operations are identified:

- Compulsory IO. This is the reading in of data files to be processed and saving the results of the computation for future use.
- Checkpointing IO. Saving the intermediate state of the program to enable a restart if the computation is interrupted for any reason.
- Out-of-core methods. Computations that are just too large to fit fully in main memory.

A set of applications covering many scientific fields has been selected and subject to investigation to better understand the requirements for high performance IO. As well as analysis of these codes to see what they do currently, attempts are being made to understand if these IO patterns are what the code developer really needs, or if they are an artifact of what gives the best performance on current architectures. Some typical performance analysis results are given in [14] and [15]. The applications covered include:

- Electron scattering for the study of low energy electron-molecule collisions.
- Terrain rendering using data from Earth observation satellites.
- Hartree Fock *ab initio* calculations.

The analysis of these codes highlights the need for improved parallel IO. For example, in the Hartree Fock calculations there is a trade off between precomputing certain integrals and computing them of the fly each time they are required. IO performance on the system used (Intel Paragon) was not sufficient to make precomputation efficient for large problems.

It was also found that the user codes used a wide range of IO request sizes, including a lot of small requests. Since IO systems give better response to large IO requests this highlights the need for libraries or parallel file systems that can efficiently service small IO requests. This could be done by caching or prefetching files in the parallel file system or by aggregating small requests through a library such as MPI-IO.

The wide variety of IO patterns observed in the applications studied suggested that it would be unwise to use a single benchmark kernel to try and measure the IO performance of a parallel file system. Also apparent was the need for a portable and standard way to use parallel IO if it is to be widely adopted by application developers. In many cases the current practice was to gather all the IO data onto a single node and perform the required file operations there. While this was a portable solution it is unlikely to offer the best scalability.

A set of IO benchmark codes has been developed within the SIO project based on some of the codes they have been evaluating. Most of these benchmarks are only available to project members, though some are in the public domain. The set of benchmarks is listed in Table 2. These give an indication of the range of applications in which high performance IO is important. A few of the publicly available ones are described in more detail in the following sections.

Application	Obtained From	Platforms	Type of IO
Astrophysics	Andrea Malagoli, Univ. of Chicago	Portable	Writes distributed arrays for checkpointing
Weather Modelling (MPMM)	John Michalakes, Argonne	Portable	Reads input data and writes history files
Climate Modelling (PCCM2)	John Michalakes, Argonne	Portable	Reads input data and writes history, restart files
Parallel Rendering	Peggy Li, JPL	Paragon, T3D	Reads initial data set and view coordinates, writes rendered frames
Seismic	Publicly available from PARKBENCH benchmarks suite	Portable	2D arrays are read/written from/to HDF style files.
Shallow Water (PSTSWM)	Ian Foster, Argonne	Portable but IO on SP/PIOFS	Checkpointing
Electronic Structure	Mike Minkoff, Argonne	CMFortran code for the CM-5	Checkpointing
Hartree-Fock	Publicly available from the Pablo web site	Portable	Creates integrals, writes them to disk, and reads them later
LU Factorization	Publicly available: Dartmouth Parallel IO Archive	nCUBE/2	Out-of-core
CFS3D	Publicly available: Dartmouth Parallel IO Archive	iPSC/x with CFS file system	Writes a distributed 3D array to a single file
BTIO Benchmark	Publicly available from NASA Ames	Portable	CFD solver periodically writes solution

Table 2: List of IO benchmarks used by the SIO project.

As well as characterisation and measurement of parallel IO applications, SIO is closely involved in the development of libraries such as ROMIO (for MPI-IO), the MPI2 standard and many other areas of parallel IO. More details about the SIO are available at:
`http://www.cacr.caltech.edu/SIO`.

3.2 The BTIO Benchmark

As part of the NAS parallel benchmark set an IO benchmark has been developed which is based on one of the computational kernels. The BT benchmark is based on a CFD code that uses an implicit algorithm to solve the 3D compressible Navier-Stokes equations. A finite difference grid is assumed and systems of 5×5 blocks at each node are solved using a block-tridiagonal solver.

The BTIO version of the benchmark uses the same computational method, but with the addition that the results must be written to disk at every fifth time step. Details of the mesh size and

results to be reported are given in [17]. The source code is available from the NAS web site and currently comes with a set of example implementations. These are:

- BT-epio: In this version each node writes its part of the solution to a local file on the local processor. This is the "embarrassingly parallel" version, which is for reference and not part of the benchmark proper.
- BTIO-fortran-direct: The BTIO benchmark requires that the results from all nodes should be written to a single file. This version uses a Fortran direct access file and allows each node to write to its own data areas concurrently.
- BTIO-simple-mpiio: Uses basic MPI-2 IO to perform the file operations.
- BTIO-full-mpiio: Again using MPI-2 IO, but with use of collective file operations.

The MPI-2 IO library and collective IO are discussed in more detail in Appendix A, but for the BTIO benchmark they provide a standard way in which data from many processors can be written efficiently to a single file. Some performance figures are given in [16] for the BTIO benchmark, using the ROMIO implementation of MPI-IO, on a range of parallel machines. A few typical results from this paper are listed in Table 3.

Machine	Processors	Unix-style	Collective
HP Exemplar	64	0.86	29.7
IBM SP	64	2.21	38.6
Intel Paragon	256	1.37	98.8
NEC SX-4	9	0.99	591.0
SGI Origin2000	36	7.93	67.2

Table 3: Data transfer rates for the BTIO benchmark. Results are write speeds as MB/s. In "Unix-style" IO each processor independently writes its part of the solution. "Collective" results use a collective MPI-IO call made by all processors.

These results illustrate the advantage of using MPI-IO and collective IO, with the BTIO benchmark. The collective IO rates are one to two orders of magnitude better than using standard Unix style IO. The BTIO benchmark URL (which was not working when we tried) is:
`http://parallel.nas.nasa.gov/MPI-IO/btio/btio-download.html`.

3.3 Parkbench SEIS1 Benchmark

As part of the Parkbench [20] suite a seismic data analysis application has been defined which includes significant IO. They refer to this as one of their "compact applications" since it is closer to the full application rather than just a computational kernel.

This benchmark performs seismic analysis of data using FFTs and related techniques. While the computation is quite easily parallelised, the amount of data to be processed can be very large and the benchmark includes test cases with up to 100GB of data files. The data processing requires a mixture of local storage for each processor element and a global results file.

This benchmark seems to have been defined late on in the Parkbench project and no results are listed on their web site. However, the benchmark has been used as a tool to evaluate new parallel file systems, such as in the Scalable IO Initiative and in the ParFiSys parallel file system [21].

The source code for this application is available on the Parkbench web site at:
`http://www.netlib.org/parkbench/future-compapps`.

4 Commercial Parallel File systems

Manufacturers of high end computing systems have already had to address some of the problems associated with the IO demands presented by large scale computations. Some of the more significant efforts in this area are outlined below. As these are commercial products there are, in general, only available on one specific type of hardware. Hence any application that is tuned to take advantage of one such solution may not be easily portable to another architecture.

4.1 PIOFS and GPFS for the IBM SP2

For their SP2 parallel architecture IBM originally provided PIOFS, a high performance parallel file system. This is now being superseded by the General Parallel File System (GPFS) [6]. GPFS allows each node of the SP2 to access files that may be spread over several IO nodes. The file system also attempts to load balance the IO activity across all disks. The SP switch is used for communication to provide fast data access. A token system is used to ensure consistent access even when concurrent reads and writes are being made to a single file.

GPFS allows efficient access using the standard UNIX file system API so that existing codes can take advantage of it without modification. The new file system is more robust than the previous PIOFS in terms of recoverability.

More details of GPFS are available at:
`http://www.rs6000.ibm.com/software/sp_products/gpfs.html`.

4.2 SGI XFS

SGI has developed the XFS [7] file system for use on all its current hardware. It has also recently made the source code publicly available and a beta version for Linux systems now exists. This code will be included in future Open Source systems such as Linux. XFS addresses many of the problems with standard Unix file systems, such as limitations on file and disk size, which are been encountered by high performance machines. However it also tries to allow high performance where possible and permits the use of parallel operations on the file system. The file system also attempts to allocate file space in contiguous areas that is necessary to achieve the fastest possible transfer rates.

Sustained performance of more than 300MB/s has been reported for SGI Challenge systems with XFS. More details are available at:
`http://www.sgi.com/software/xfs/overview.html`

4.3 SUN PFS

As part of their HPC ClusterTools V3.1, Sun include the Sun Parallel File System. The source code to HPC ClusterTools is available to developers under the Sun Community Source Code License, which gives free access for non-commercial use of the software. Sun PFS is described as:

> The Sun Parallel File System (PFS) supports high-performance, scalable IO, distributing individual files across multiple disks and servers. PFS is integrated with the Solaris operating environment by means of a kernel module that implements the virtual file system interface. PFS can span multiple storage systems, whether

attached to a single node or to multiple nodes; for example, a file system can be constructed to span all storage devices within a cluster. PFS allows parallel applications to perform high-performance, scalable IO by moving data among multiple storage systems and multiple processes in parallel. Sun Parallel File system is completely complementary to MPI IO to achieve truly sustainable parallel IO throughput in production type environments.

More details are available at:
`http://www.sun.com/software/solutions/hpc/docs/index.html`

4.4 Intel Paragon PFS

The Intel Paragon provides IO via a set of IO nodes with appropriate disk or RAID devices attached. Intel also provides PFS, a Parallel File System [8], to allow all nodes to read and write files that are striped across the IO nodes. Various access modes are supported to allow separate processes to read and write files either independently or collectively. Asynchronous IO is also supported.

PFS was also used in the newer TFLOPS supercomputer from Intel. Transfer rates of up to 1GB/s are reported. More details are given at:
`http://developer.intel.com/technology/itj/q11998/articles/art_3.htm`

4.5 Cray T3E File system

The Cray T3E supports high performance file systems based on striping files across multiple disks. A recent review of the performance of T3E IO is given in [10]. The IO nodes of the T3E are connected to the processor nodes via a "GigaRing" connection. This allows all processors to have equal access to IO devices. The file system has some support for global access to a single file (which may be striped across disks) from all processors. Until recently this was restricted to Cray specific system calls. However, support for a standard interface to parallel IO, through the addition of MPI-IO (based on ROMIO) is now available in the Cray version of MPI (MPT).

Tests in [10], show that IO rates of around 100MB/s are achievable from a single processor writing 64MB of contiguous data striped across 10 disks. Results writing a global file, accessed from all processors using Fortran direct access, also showed a transfer rate of about 100MB/s for the T3E system they used. No tests were made using MPI-IO in [10] as it was not available at the time. Cray state that the largest T3E systems can have an IO bandwidth up to 128GB/s. More details are available at:
`http://www.cray.com/products/systems/crayt3e/scalableio.html`

4.6 NEC SFS

On the SX4 and SX5 range of supercomputers NEC provides the Supercomputing File System (SFS). This supports single files up to 2 TB size and allows use of large disk arrays to support this. A flexible caching scheme is used allowing control of various file system parameters. For large data movements caching can be avoided altogether to give better performance.

SFS allows large clusters of disk space to be allocated to a file to ensure high transfer rates. On the SX4 up to 4 IO processors can be attached to each compute node and these can perform asynchronous IO in parallel. The figures in Table 3 show the high performance levels possible using MPI-IO on an SX-4 system.

4.7 Hewlett-Packard Exemplar File System

Hewlett-Packard provides a High Performance File System (HFS) and a Journaling File System (JFS) on its high end SMP server machines. Some experimental measurements of an Exemplar file system (circa 1997) are given in [32]. The file system supports very large files (128GB) and allows several extensions to the standard Unix API, such as vectored read and write operations for efficient non-contiguous data access. Parallel write performance of 2.7GB/s is reported using 64 processors to write a single 128MB file. This level of performance must be due to caching, since the file system used was striped across 12 disks capable of 10MB/s each.

4.8 Compaq TruCluster File System

Compaq market high performance clusters such as the GS320 providing 32 Alpha processors running Tru64 UNIX. Three file systems are supported, NFS, CFS and PFS. The main file system is CFS (Cluster File System) which provides a uniform file name space across all processors in the cluster. For high performance file access from a single application, the PFS (Parallel File System) can be used. This allows a file system to be striped across multiple physical file systems and also allows the user to fine tune aspects, such as the strip size, using system calls. The physical file systems themselves can use RAID storage. Support for MPI-IO, based on the ROMIO package, is also provided under PFS. More details are available at Compaq's web site, see for example:
`http://www.compaq.com/alphaserver/download/scseriesv1.pdf`

5 Other Parallel File System Projects

Apart from SGI's recent release of XFS, a number of projects have tried to develop platform independent parallel file systems. Some of these have been released under licences such as GPL or LGPL. Others are available for research use only, while others are still under development. A few of these are briefly described here.

5.1 PVFS

PVFS is a Parallel Virtual File System for Linux clusters developed by Carns [9]. Like some of the commercial systems listed above, PVFS assumes that a set of nodes within the cluster will be designated as IO nodes and have suitable disks (or RAID devices) attached to them. User files can then be striped across these disks to obtain high performance. Three interfaces are currently provided to the file system:

- A native API that accesses the file system functions directly.
- The MPI-IO API (see section 6.1) allowing standard MPI-IO programs to obtain high performance.
- UNIX (POSIX) API, so that existing programs can use PVFS. This depends on use of a dynamic library to intercept the standard IO system calls.

Using a Myrinet cluster and MPI-IO with PVFS, read and write bandwidths of 700 MB/s have been reported in [9] using a cluster with 32 IO nodes each with a single 9GB disk attached.

PVFS version 1.4.4 was released in June 2000 and the web site indicates that developments are continuing. More details are available at:
`http://parlweb.parl.clemson.edu/pvfs`

5.2 PPFS

PPFS is a Portable Parallel File System that has been developed by members of the Scalable IO Initiative [18]. It is described as:

> PPFS, the Portable Parallel File System, is a parallel IO library invoked by user IO calls that offers control over a variety of input/output configurations such as disk striping factor, prefetch and caching policies. It was developed to explore the design space for scalable IO systems, and relies on the underlying system software for physical IO.

The software runs on many machines including the Convex (HP) Exemplar, IBM SP, Intel Paragon, SGI Power Challenge and Sun Solaris clusters. It requires an existing message passing library, such as MPI, PVM or NX to be present. PPFS provides facilities such as file caching and dynamically reconfigurable writebehind and prefetching.

A new version of this file system, PPFS-II, is currently under development. It is based on the idea that since applications have such widely differing IO requirements a parallel file system must provide real time control and adaptive policies for caching and prefetching files. Fuzzy logic rules are used to select the most appropriate striping size based on IO contention and available storage. PPFS-II also makes use of some of the Globus [13] distributed computing infrastructure. Major performance improvements are claimed using these optimisations.

PPFS V1.0 is not public domain but is available for research and non-profit use. PPFS-II is still under development and does not appear to be available. More details about PPFS and PPFS-II are available at:
`http://www-pablo.cs.uiuc.edu/Software/PPFS/ppfs.htm`

5.3 PIOUS

PIOUS has been developed at Emory University as a portable parallel IO system for use with PVM3. The documentation describes it as:

> Just as PVM implements a virtual multicomputer on top of a heterogeneous network of computing resources, PIOUS implements a fully functional parallel file system on top of PVM. PVM applications obtain transparent access to shared permanent storage via PIOUS library functions.
>
> PIOUS is intended to serve both as a platform for supporting high-performance parallel applications, and as a vehicle for parallel file system research. PIOUS implements the traditional functionality found in most parallel computer file systems, as well as a number of unique features, including:
>
> - two-dimensional file objects and logical file views
> - coordinated file access with guaranteed consistency semantics
> - data declustering for scalable performance
> - transaction support and user-selectable fault tolerance modes

- extended file maintenance primitives for managing declustered files
- C and Fortran language bindings.

PIOUS should work on most systems that support PVM 3. To date, PIOUS has been tested on the following: Sun4/SunOS 4.1.3/5.3-4, SGI/IRIX 4.0.5/5.3, Dec Alpha/OSF1 2.1, HP 9000/HP-UX, and IBM RS6000/AIX.

PIOUS is released under the GNU LGPL and the latest version appears to be 1.2.2 released in 1995. More details are available at:
`http://www.mathcs.emory.edu/pious`

5.4 GFS: The Global File System

GFS is another file system design for use with Linux clusters by Sistina Software. They described it as:

> The Global File System (GFS) [27] is a shared disk cluster file system for Linux. GFS supports journaling and recovery from client failures. GFS cluster nodes physically share the same storage by means of Fibre Channel or shared SCSI devices. The file system appears to be local on each node and GFS synchronises file access across the cluster. GFS is fully symmetric, that is, all nodes are equal and there is no server that may be a bottleneck or single point of failure. GFS uses read and write caching while maintaining full UNIX file system semantics.

This file system is still under active development and version 4.0beta has recently been released. The software is freely available under the Gnu Public License. In [27] read and write rates of 55MB/s are shown for a single Alpha processor node connected to a storage array consisting of 8 fibre channel disk drives. Scalability is said to be good, though tests only extend to 8 compute nodes. More details are available at:
`http://gfs.lcse.umn.edu/`

5.5 ParFiSys

The ParFiSys [21] file system was developed as part of an Esprit project at Universidad Politécnica de Madrid (UPM), Spain, and was originally known as the Cache Coherence File System (CCFS). The system is described as:

> The main goals of ParFiSys are to provide IO services to scientific applications requiring high IO bandwidth, to minimise application porting effort, and to exploit the parallelism inherent to generic message-passing multicomputers, including processing nodes (PN) and IO nodes (ION). ParFiSys is being used to explore a variety of data distribution, distributed caching and prefetching strategies: disk data striping factors, file and disk block prefetch policies, caching policies, cache coherence models, and IO patterns.

The file system was originally developed for Sun clusters, T800 and GPMIND machines, and an IBM-SP2 implementation is also mentioned. Recent research [22] has been into caching files across both IO and compute nodes with aggressive prefetching and delayed-write techniques. The software is described as a research code but is available under GPL terms. The latest

version is 1.2 which appears to date from 1996 and does not support the SP2. More details are available at:
`http://laurel.datsi.fi.upm.es/~gp/parfisys.html`

5.6 The Hurricane File System

The Hurricane File System (HFS) [23] is another portable file system for SMP systems. It is described as:

> The Hurricane File System is developed for large-scale shared memory multiprocessors. Since application IO requirements for large-scale multiprocessors are not yet well understood, the file system must be flexible so that it can be extended to support new IO interfaces. The main goals are scalability, flexibility and to maximise IO performance for a large set of parallel applications. There are three important features of HFS:
>
> - Flexibility: HFS can support a wide range of file structures and file system policies.
> - Customizability: The structure of a file and the policies invoked by the file system can be controlled by the application.
> - Efficiency: little IO and CPU overhead

HFS runs on various systems including IBM AIX, SunOS and SGI IRIX. The source code does not appear to be available at the web site, and it is not clear that this project is still active. More details are available at:
`http://www.eecg.toronto.edu/parallel/hurricane.html`

5.7 The Galley Parallel File System

The Galley Parallel file [24] system is an experimental file system that was developed by Nils A. Nieuwejaar at Dartmouth. The source code has been released, but appears to be unsupported. It runs on IBM-SP2 and RS6000 workstation clusters. More details are available at:
`http://www.cs.dartmouth.edu/~nils/galley.html`

5.8 HiDIOS Parallel File System

The HiDIOS Parallel File System was developed at the Australian National University for the Fujitsu AP1000 parallel machine. It supports the normal Unix file system API, so existing codes can easily make use of it. Data transfer rates of up to 40MB/s are reported. The web page was last updated in 1995. More details are available at:
`http://cafe.anu.edu.au/cap/projects/filesys`

6 Parallel IO Libraries

There is a lot of interest in development of interfaces that allow the application programmer to efficiently use the underlying file system from a parallel machine. Ideally such interfaces should be both portable and standardised if it is to achieve widespread acceptance. The distinction between a library to support parallel IO and a parallel file system is not well defined and we have followed the statements made by the respective projects in this respect.

Perhaps the most notable effort in this area is the MPI-IO standard that has now been incorporated into the latest version of MPI, MPI-2. This standard, and the available implementations, are discussed here along with some other library projects. Some basic examples of the use of MPI-IO are given in Appendix A.

6.1 MPI-IO

One of the major problems in trying to develop software using parallel IO is that there has, until recently, been no standard way to access high performance file systems. This has changed with the introduction of the MPI-IO that now forms part of MPI-2 standard [4].

MPI-IO provides an interface via which the programmer can request collective IO operations over all processors working on a job. For example, all processors in a mesh based computation may need to write their current part of the solution to the appropriate part of a single shared output file. One way to do this in Fortran would be to use a direct access file that each processor opens independently and writes its results into, using an appropriate offset. Even if the operating system permits several processors writing a file in this way, the performance may be very poor as the separate blocks of data are written in a random order, causing a lot of movement of the disk write heads.

Using an MPI-IO collective operation to write the data from all processors to the disk file permits the system to order the data into contiguous blocks and hence maximise the disk data transfer rate. This illustrates one of the basic ways in which MPI-IO can help enhance performance. Other important features provided by MPI-2 include a standardised way to access asynchronous file IO, permitting the overlap of computation and IO, and optimisations for non-contiguous data access.

Some implementations of MPI-IO are now available. The most notable ones are discussed below. The MPI-2 standard, including the MPI-IO chapter, is available at:
`http://www.mpi-forum.org`

6.1.1 ROMIO

The ROMIO [1] [16] implementation of MPI-IO appears to be the most widely used version at present. It is freely available and version 1.0.3 was released in September 2000.

The software is portable and is described as:

> ROMIO runs on at least the following machines: IBM SP; Intel Paragon; HP Exemplar; SGI Origin2000; Cray T3E; NEC SX-4; other symmetric multiprocessors from HP, SGI, DEC, Sun, and IBM; and networks of workstations (Sun, SGI, HP,

[1] The name is thought not to be an acronym, but is related to the fact that the developer previously wrote the PASSION file system.

IBM, DEC, Linux, and FreeBSD). Supported file systems are IBM PIOFS, Intel PFS, HP/Convex HFS, SGI XFS, NEC SFS, PVFS, NFS, and any Unix file system (UFS).

ROMIO is optimised for noncontiguous access patterns, which are common in parallel applications. It has an optimised implementation of collective IO, an important optimisation in parallel IO. ROMIO 1.0.3 includes everything defined in the MPI-2 IO chapter except support for file interoperability and user-defined error handlers for files.

C, Fortran, and profiling interfaces are provided for all functions that have been implemented. We have implemented the subarray and distributed array datatype constructors from the MPI-2 miscellaneous chapter, which facilitate IO involving arrays. We have also implemented the info functions from the MPI-2 misc. chapter, which allow users to pass hints to the implementation.

ROMIO is designed to be used with any MPI implementation. It is, in fact, included as part of several MPI implementations: The latest version, 1.0.3, is included in MPICH 1.2.1; an earlier version is included in LAM, HP MPI, SGI MPI, and NEC MPI.

The standard ROMIO distribution includes some test programs that illustrate the use of MPI-IO. Some of these examples are discussed in Appendix A.

As has already been mentioned, the results in [16] show that this library can give substantial benefits for parallel IO performance though the use of data sieving and collective operations. More details of ROMIO are available at: `http://www-unix.mcs.anl.gov/romio`

6.1.2 Other MPI-IO implementations

Many hardware manufactures are now offering their own MPI-IO implementations as they move to support the MPI-2 standard. Some of these are actually directly based on the portable ROMIO software. Sun provide an MPI-IO implementation that is integrated with their HPC Cluster Tools product and works with the Sun Parallel File System.

The HPSS product (see section 7.7) also includes its own MPI-IO API for users to access data. This is in addition to various other APIs, such as parallel ftp and NFS protocols.

6.2 TPIE

The Transparent Parallel IO Environment (TPIE) project addresses the problem of out-of-core algorithms that are needed when a problem is too large to be contained in main memory. The software is described as:

> The goal of theoretical work in the area of external memory algorithms (also called IO algorithms or out-of-core algorithms) has been to develop algorithms that minimise the Input/Output communication (or just IO) performed when solving a given problem. The area was effectively started in the late eighties by Aggarwal and Vitter and subsequently IO algorithms have been developed for several problem domains. Also IO performance can often be improved if many disks can efficiently be used in parallel and the use of parallel disks has received a lot of theoretical attention.
>
> TPIE is designed to bridge the gap between the theory and practice of parallel IO systems. It is intended to demonstrate all of the following simultaneously:

- Abstract away the details of how IO is performed so that programmers need only deal with a simple high level interface.
- Implement IO-optimal paradigms for large scale computation that are efficient not only in theory, but also in practice.
- Remain flexible, allowing programmers to specify the functional details of computation taking place within the supported paradigms. This will allow a wide variety of algorithms to be implemented within the system.
- Be portable across a variety of hardware platforms.
- Be extensible, so that new features can easily be added later.

TPIE is implemented as a set of templated classes and functions in C++. It also includes a small library and a set of test and sample applications.

The software is still under development and is available under the GPL. The current version is 0.9.01b released in November 1999. More details are available at:
`http://www.cs.duke.edu/TPIE/`

6.3 ChemIO: Parallel IO for Computational Chemistry

This work is a joint effort between Argonne and Pacific Northwest laboratories. It is also related to the SIO and is focused on parallel IO for chemistry codes. The ChemIO library [28] is designed to provide a portable API for high performance IO for massively parallel computations. The library offers three types of storage:

- Disk Resident Arrays(DRA), an extension of global arrays to secondary storage.
- Exclusive Access Files (EAF) - per-processor private files.
- Shared Files (SF): files to which multiple processors can read and write independently.

ChemIO is implemented in a modular fashion. The user level libraries (DRA, EAF, SF) are layered upon the ELIO (Elementary IO) device library that provides a portable interface to different file systems. It is planned to allow ELIO to interface to MPI-IO libraries, to enhance its portability.

The library supports asynchronous IO to overlap communication and computation and has been successfully used with several codes including the semi-direct SCF (Self Consistent Field, or Hartree Fock) method implemented in NWChem. More details are available at:
`http://www.mcs.anl.gov/chemio/`

6.4 The Panda Project

This project is based at the Database Research Laboratory, University of Illinois. The aims are described as:

> Our goal is to produce new data management techniques for IO intensive applications that will be useful in high-performance scientific computing. Specifically, we are examining the problem of efficient support for applications that make use of extremely large multidimensional arrays on secondary storage. We have three goals:

to provide simpler, more abstract interfaces to application programmers, to produce more advanced IO libraries supporting efficient layout alternatives for multidimensional arrays on disk and in main memory, and to support high performance array IO operations.

The original Panda file system [29] is still under active development with version 2.1 released in 1996 and 3.0alpha in 1999. The terms of the license are unclear, but the software appears to be freely available.

The library is written in C++ and the system has been tested on an IBM SP2 and clusters of Sun and HP workstations. Techniques such as collective IO are used by the library. More details are available at:
`http://cdr.cs.uiuc.edu/CDR/panda`

6.5 PASSION

PASSION, Parallel And Scalable Software for Input-Output [19], is targeted at supporting IO intensive out-of-core solution methods. It consists of a compiler and a runtime support library. The runtime libraries provide efficient IO operations that can be used by the compiler. Input to the complier is typically in a data parallel language, such as HPF, which is converted to node programs using the runtime library. A parallel file system (VIP-FS) is also included in the PASSION distribution. There are not many references to PASSION since 1995 and web links to the University of Syracuse PASSION site were broken when we tried them.

6.6 ViPIOS

ViPIOS [25], the Vienna Parallel Input-Output System, takes an approach that combines ideas from parallel IO libraries with those from parallel file systems. The application program makes calls to the ViPIOS library for all IO requests. These are then passed to a server process (of which there can be many) which in turn decides what requests to make to the underlying file system.

The initial objective of this system was to support programs written in High Performance Fortran, such as VFC, the HPF derivative of Vienna Fortran. Some basic support of the MPI-IO interface is also provided. More details are available at:
`http://www.unet.univie.ac.at/~a8926464/research.htm`

6.7 Jovian parallel IO library

This project was based at the University of Maryland and is related to both Chaos and SIO. The driving force was large scale problems such as Grand Challenge project on Land Cover Dynamics, processing vast amounts of remote sensing data. A major theme was compiler based optimisation of data placement for parallel IO beyond the limits of prefetch and write-behind of the operating system. Most recent publications are 1996. More details are available at:
`http://www.cs.umd.edu/projects/hpsl/io/io.html`

7 Other Parallel IO Related Projects

A number of other important projects related to various aspects of parallel IO have been identified. These are listed here, along with URLs that give more details of the projects.

7.1 RIO - Remote IO for Metasystems

This is related to the Globus project on Grid based computing [13]. The software is described as:

> The RIO library provides basic mechanisms for tools and applications that require high-performance access to data located in remote, potentially parallel file systems. RIO implements the ADIO abstract IO device interface specification, which defines basic IO functionality that can be used to implement a variety of higher-level IO libraries.

Since ROMIO is based on the ADIO library this tool should enable efficient MPI-IO access to remote file systems. With suitable high speed network connections this will allow large remote data sets to read and written without the need to store them on the local file system.

- URL `http://www.globus.org/details/rio.html`

7.2 Argonne scalable IO research

The Argonne National Laboratory is involved in many aspects of parallel IO and is closely associated with the Scalable IO initiative. ROMIO and ADIO, the abstract device interface on which ROMIO is built, have been developed here. They are also working on the IO characterisation and application test suites for the SIO. Other interests include RIO, Remote IO for metacomputing systems, the ChemIO project and multi-threading for scalable IO.

- URL `http://www.mcs.anl.gov/scalable/scalable.html`

7.3 The CHAOS project

Based at the University of Maryland, the Chaos project has studied many aspects of parallel data intensive computing. Recent work has been related to filtering very large scientific data sets held on archival storage [26] for Grid type applications.

- URL `http://www.cs.umd.edu/projects/hpsl/Chaos.htm`

7.4 Dartmouth research in parallel IO

An extensive bibliography of parallel IO related research is available at the Dartmouth College site.

- URL `http://www.cs.dartmouth.edu/research/pario.html`

7.5 Gemini storage systems laboratory

Research into efficient RAID disk arrays and clusters is carried out at the Gemini Laboratory.

- URL `http://www-cse.ucsd.edu/groups/gemini`

7.6 GigaServer: Parallel Image Storage and Processing Servers

This project, based at the Peripheral Systems Laboratory of EPFL in Switzerland, provides a framework to build high performance data servers from networks of WindowsNT PCs. Fast Ethernet (or Gigabit Ethernet) is used along with high speed SCSI disks. This work appears to follow on from an earlier project, PS^2. A major demonstrator is a server for a high resolution volume representation of a human body. With a data set of about 13GB, an 8 processor, 60 disk system is able extract a requested slice using data rates of 108MB/s.

Another project at EPFL is SFIO, a striped file IO library for parallel IO in an MPI environment. This library can be used either directly, or in the future via an MPI-IO interface. Future extensions will support collective IO, non-blocking operations and other disk optimisations to provide a high performance MPI-IO interface.

- URL `http://diwww.epfl.ch/w3lsp/pub/research/`

7.7 HPSS – High-Performance Storage System

This large and on-going project is described as:

> The High Performance Storage System (HPSS) is software that provides hierarchical storage management and services for very large storage environments. HPSS may be of interest in situations having present or future scalability requirements that are very demanding in terms of total storage capacity, file sizes, data rates, number of objects stored, and number of users. HPSS is part of an open, distributed environment based on The Open Group's Distributed Computing Environment (DCE) products that form the infrastructure of HPSS. HPSS is the result of a collaborative effort by leading US Government supercomputer laboratories and industry to address very real, very urgent high-end storage requirements. HPSS is offered commercially by IBM Global Government Industry, Houston, Texas.

To provide high performance IO storage devices are connected directly to a high speed network. A HPSS server controls the movement of data between these devices and the compute nodes, but the data itself flows directly from the disks to the requesting node, so avoiding the bottleneck of a single server. A disk array device can provide over 50MB/s to a node and the architecture allows parallel use of such devices. Support of MPI-IO is planned for future releases. The initial implementation was based on IBM RS6000 systems, but the code is portable to any Unix system and Sun have recently announced HPSS support on their systems.

- URL `http://www.sdsc.edu/hpss`

7.8 Network-Attached Peripheral

This project, based at Lawrence Livermore National Laboratory, was part of the Scalable IO Facility (SIOF), and is also related to the work on HPSS (High-Performance Storage Systems). A Network Attached Peripheral (NAP) is envisioned as being a device such as a disk, RAID array or tape that is available on the network, rather than been dedicated to a single processor. The project sought to promote the production of cheap commodity interfaces to such devices that could be used efficiently on MPP systems. Data flow would be control as part of HPSS. The web pages on NAP were last updated in 1995.

- URL `http://www.llnl.gov/liv_comp/siof/siof_nap.html`

7.9 Parallel IO Project at LLNL

This work was another project at Lawrence Livermore related to the SIOF. A lot of effort was related to the development and implementation of the MPI-IO standard. They also investigated ways to use MPI-IO with interfaces to netCDF and HDF data formats. Work is underway to produce a standard test suite for MPI-IO and to test it with real applications across various platforms. Web pages last updated 1998.

- URL `http://www.llnl.gov/sccd/lc/piop`

7.10 Parallel Data Laboratory

The Parallel Data Laboratory (PDL) runs a number of projects related to aspects of parallel IO. These include Active Disks, network attached storage devices, Abacus, a tool to dynamically determine the best placement of data for a cluster of workstations and TIP, Transparent Informed Prefetching and Caching. These projects appear to be still active.

The Scotch Parallel Storage Systems [30] project was another research effort based at the PDL. Its main aims were to develop more efficient RAID devices and ways in which they can best be integrated into parallel file systems. This works dates from 1995.

- URL `http://www.pdl.cmu.edu/`

7.11 RAID papers

Set of papers on RAID technology. Not changed since 1994.

- URL `ftp://ftp.cs.berkeley.edu/ucb/raid/papers`

7.12 RAPID-Transit project

This project ran from 1987-91 and is described as:

> RAPID-Transit was a testbed for experimenting with caching and prefetching algorithms in parallel file systems (RAPID means "Read-Ahead for Parallel Independent Disks"), and was part of the larger NUMAtic project at Duke University. The

testbed ran on Duke's 64-processor Butterfly GP1000. The model we used had a disk attached to every processor, and that each file was striped across all disks. Of course, Duke's GP1000 had only one real disk, so our testbed simulated its disks. The implementation and some of the policies were dependent on the shared-memory nature of the machine; for example, there was a single shared file cache accessible to all processors. We found several policies that were successful at prefetching in a variety of parallel file-access patterns.

- URL `http://www.cs.dartmouth.edu/~dfk/rapid-transit.html`

7.13 ViC*, a virtual-memory C*

This work at Dartmouth is described as:

> ViC* is a system for data-parallel computing with parallel data so large that it does not fit in main memory. Although one could simply use the native demand paging found on most systems for such "out-of-core" problems, ViC* is able to take advantage of the structure within data-parallel programs and architectural advances such as parallel disk systems to deliver significantly better performance.
>
> The ViC* system consists of a compiler and a run-time system. The compiler translates C* programs with shapes declared out of core, which describe parallel data stored on disk. The compiler output is a SPMD-style program in standard C with IO and library calls added to efficiently access out-of-core parallel data. The ViC* compiler also applies several program transformations to improve out-of-core data layout and access. The run-time system performs parallel disk accesses and invokes optimal out-of-core algorithms developed for the Parallel Disk Model (PDM) when appropriate.

The software is not on the web site, but is available on request. It requires either a Unix file system or an interface to MPI-IO, such as ROMIO.

- URL `http://www.cs.dartmouth.edu/~thc/ViC.html`

8 Conclusions

There is a wide spread need for high performance IO to support the increasing computational speed of high end systems. Developments in both hardware and software are trying to satisfy these requirements.

On the hardware side magnetic disk technology is still the dominant media, along with magnetic tape backups. The physical characteristics of disks mean that they have to be used in parallel to achieve the very high transfer rates required. Other bottlenecks, such as the disk controllers and the interconnection within the machine also call for parallelism at these levels as well. With many disks operating in this way it is essential that physical IO is only performed in very large chunks to get the best data transfer rates. This requires intelligent caching and read ahead by the hardware and appropriate support from software interfaces that use these facilities.

These hardware requirements are being met by the computer manufacturers in various ways, such as the high performance parallel file systems provided IBM, Sun, Cray, etc. There is also the move to HPSS for the provision of high speed access to storage devices, that is now available for IBM, Sun and other Unix systems.

On the software front, the MPI-IO chapter of MPI-2 appears to be the most widely supported (and standardised) way to interface to high performance file systems. However, there is still some way to go here since even the most commonly used implementation (ROMIO) does not yet support all the features within the standard. Also the highest performance may require the use of "hints" can vary from one machine to another. Another advantage of MPI-IO is that libraries such as RIO within the Globus project should allow easy access to remote parallel file systems in future grid based computations.

MPI-IO is a rather low level library where the user has to specify all the details of the data placements. There is clearly room for higher level interfaces such as the ChemIO project that treats data movements in a simpler manner for the user. While several projects address this problem, there is no clear portable standard at present.

A MPI-IO Examples

A few examples of the type of parallel IO operations that are supported by the MPI-IO chapter of MPI-2 are illustrated in this appendix. The MPI-2 standard is available online at: `http://www.mpi-forum.org` and it is described in a book [33]. These examples are taken from the `test` directory that is distributed with the ROMIO implementation of MPI-IO. This package can be easily obtained on the net and is often already included in many popular implementations of MPI, such as MPICH.

Most of the ROMIO examples are provided in C though some have also been converted to Fortran 77. The conversions are not always perfect, for example one case fails when array bound checks are enabled due to the use of an index from zero at one point.

A.1 Simple example of MPI-IO

In the example `simple.c` each process opens its own file, with a unique name, and independently writes to (and reads from) it. This could be done without MPI-IO, but it serves to illustrate the basic operations. It is possible that the MPI-IO library will allow some optimizations above those available with the use of standard Fortran. For example, the way each file is opened makes it clear that it is to be used by only one processor, while this would not be evident if the standard `OPEN` statement were to be used.

In C the file related declarations and operations are given by:

```
#include <mpi.h>
...
    MPI_File fh;
    MPI_Status status;
...
    MPI_File_open(MPI_COMM_SELF, filename, MPI_MODE_CREATE | MPI_MODE_RDWR,
                  MPI_INFO_NULL, &fh);
    MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
    MPI_File_write(fh, buf, nints, MPI_INT, &status);
    MPI_File_close(&fh);

    MPI_File_open(MPI_COMM_SELF, filename, MPI_MODE_CREATE | MPI_MODE_RDWR,
                  MPI_INFO_NULL, &fh);
    MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
    MPI_File_read(fh, buf, nints, MPI_INT, &status);
    MPI_File_close(&fh);
...
```

In the above code extract an MPI file pointer, `fh`, is declared and passed to the `MPI_File_open` routine. The file open routine is a collective routine which is called by a group of processes to open a single file. In this case each process is opening its own file so the communicator `MPI_COMM_SELF` is used and the file name argument is different for each process. Note that various flags are passed to the open routine to indicate what operations are to be made on the file. The available flags are:

MPI_MODE_RDONLY read only

MPI_MODE_RDWR reading and writing

MPI_MODE_WRONLY write only

MPI_MODE_CREATE create the file if it does not exist

MPI_MODE_EXCL return error if creating file that already exists

MPI_MODE_DELETE_ON_CLOSE delete file on close

MPI_MODE_UNIQUE_OPEN file will not be concurrently opened elsewhere

MPI_MODE_SEQUENTIAL file will only be accessed sequentially

MPI_MODE_APPEND set initial position of all file pointers to end of file

This information may be useful to the underlying file system, for example to control caching policy. Other information about the file use and hints to the system can be provided via the forth option which is a pointer to the type `MPI_Info`. A null value is used above, but various options can be set, such as the number of IO nodes to use, the striping block size, etc. As these are only hints, an implementation of MPI-IO may ignore them. The success of the open operation is indicated by the return value of the function in C, or the additional argument in Fortran.

The `MPI_File_set_view` call is used to specify what data can be read from or written to the file. The parameters are the file handle (from the open call), a initial displacement into the file to start reading at (zero in this case), followed by the elementary data type and the file data type. In this case both these are type `MPI_INT` to indicate that IO will consist of integer values (the etype) and that each "unit" of the file (the filetype) is one integer. In this case the independent files are just contiguous integer streams but more complex cases are possible where each process has a "view" of a different part of the same file. The fifth argument to `MPI_File_set_view` is the data representation to use. The "native" option will use the same data format as in memory, which gives the best performance but may cause problems in heterogeneous MPI environments. The popular ROMIO library currently only supports the native format but the standard also specifies an "external32" format which should allow interoperability between different machines and MPI versions.

The write operation is then performed by `MPI_File_write` which just sends the requested number of integers (`ninit`) from the buffer to the file. This is the simplest MPI write routine and there are many others which cover asynchronous and collective IO. Note that the user must close the files using `MPI_File_close` before calling `MPI_Finalize`.

A.2 Collective IO

Collective IO operations can be useful when several processors need to write a single file. In the past this problem has often been addressed by sending all the data to a single process that assembles it and performs the IO. However this is not scalable when there are many processors and the data is to be striped across several IO nodes.

The ROMIO example file `fcoll_test.f` illustrates how an array that is block distributed across all processors may be written with MPI-IO. The test program assumes an array size of $32 \times 32 \times 32$ and if we have say 8 processors then each will get one quadrant of the three dimensional array.

Each MPI process first creates a new data type to describe how its local array maps to the global array. This is done with the following code in Fortran, using array notation for compactness, though the original example is Fortran 77:

```
   include 'mpif.h'
...
   ndims = 3
   order = MPI_ORDER_FORTRAN

   array_of_gsizes(1:3) = 32
   array_of_distribs(1:3) = MPI_DISTRIBUTE_BLOCK
   array_of_dargs(1:3) = MPI_DISTRIBUTE_DFLT_DARG
   array_of_psizes(1:3) = 0

   call MPI_DIMS_CREATE(nprocs, ndims, array_of_psizes, ierr)
   call MPI_TYPE_CREATE_DARRAY(nprocs, mynod, ndims,
  &    array_of_gsizes, array_of_distribs, array_of_dargs,
  &    array_of_psizes, order, MPI_INTEGER, newtype, ierr)

   call MPI_TYPE_COMMIT(newtype, ierr)
```

An integer array `writebuf` represents the $32 \times 32 \times 32$ data array which is to be blockwise distributed across the available `nprocs` processors. The newly defined MPI type `newtype` will represent the local data and its mapping to the global array. The first MPI routine, `MPI_DIMS_CREATE`, finds a suitable decomposition for the available number of processors, e.g. with 8 processors in 3D it gives a $2 \times 2 \times 2$ processor array. The second call, `MPI_TYPE_CREATE_DARRAY`, creates the data type which is then committed.

In the example program the local buffer is then populated with some test data before being written to and read back from a single global file using collective IO operations. The write operation is as follows:

```
   call MPI_FILE_OPEN(MPI_COMM_WORLD, str,
  &       MPI_MODE_CREATE+MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierr)
   disp = 0
   call MPI_FILE_SET_VIEW(fh, disp, MPI_INTEGER, newtype, 'native',
  &    MPI_INFO_NULL, ierr)
   call MPI_FILE_WRITE_ALL(fh, writebuf, bufcount, MPI_INTEGER,
  &    status, ierr)
   call MPI_FILE_CLOSE(fh, ierr)
```

In this case the file open operation is a collective one carried out by all the processes. The previously defined data type `newtype` is now used as the "file type" argument in setting the file view. This informs MPI that the local buffer should be mapped to the appropriate locations in the output file and as each process has a different `newtype` according to its rank, each will write separate areas of the file. The actual data transfer is made by the `MPI_FILE_WRITE_ALL` call. This is also a collective operation across all processors. The MPI-IO implementation may achieve extra efficiency through combining data into a few large (and contiguous) writes rather than many small ones.

A.3 Asynchronous IO

Asynchronous IO operations are supported in the new MPI-2 standard in a similar manner to the asynchronous message passing. Thus there are routines such as `MPI_File_iread` which

return immediately and provide a request handle that can be tested or waited for. A simple example of this is provided in the `async.c` where the key parts of the code are:

```
  MPI_File_open(MPI_COMM_SELF, filename, MPI_MODE_CREATE | MPI_MODE_RDWR,
                MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
  MPI_File_iwrite(fh, buf, nints, MPI_INT, &request);
/* other computation may be included here */
  MPIO_Wait(&request, &status);
  MPI_File_close(&fh);
```

In this case each processor opens its own file (`MPI_COMM_SELF`) and starts writing a stream of integer data to it. While the IO is being performed other computations can run. `MPIO_Wait` is used to ensure that the operation is complete before closing the file.

A.4 Noncontiguous IO

If a process needs to write (or read) many small data items which are close together in a file, but not contiguous, it may be possible to perform optimizations such as reading the whole disk area in question, changing just those values to be updated, and then writing the new disk block. The example program `noncontig.c` shows one way in which MPI-IO may be used to issue a request for noncontiguous IO. It is of course up to the MPI implementation as to how efficient the operation will be.

The first write operation in the test case is based on defining a new MPI data type to describe a noncontiguous integer vector and then using this to define a structure type. This is done by the code:

```
  MPI_Type_vector(SIZE/2, 1, 2, MPI_INT, &typevec);

  b[0] = b[1] = b[2] = 1;
  d[0] = 0;
  d[1] = mynod*sizeof(int);
  d[2] = SIZE*sizeof(int);
  t[0] = MPI_LB;
  t[1] = typevec;
  t[2] = MPI_UB;

  MPI_Type_struct(3, b, d, t, &newtype);
  MPI_Type_commit(&newtype);
  MPI_Type_free(&typevec);
```

The new data type can then be used directly in a write operation such as:

```
    MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE | MPI_MODE_RDWR,
                  info, &fh);
    MPI_File_set_view(fh, 0, MPI_INT, newtype, "native", info);
    MPI_File_write(fh, buf, 1, newtype, &status);
```

Noncontiguous IO can also be combined with collective IO, as in the case where several processors need to write interleaved values into a file. The same method can be used, with each process

defining a data type that describes the values it needs to write and setting this as its file view. Then it is just necessary to replace the non-collective IO call with `MPI_File_write_all`.

References

[1] R.Oldfield and D.Kotz, "Applications of Parallel I/O", Technical Report PCS-TR98-337, Department of Computer Science, Dartmouth College, Hanover (Aug 1998).

[2] "Parallel I/O for High Performance Computing", J.M.May, Morgan Kaufmann Pub., ISBN 1-55860-664-5 (Oct 2000).

[3] P.Brezany and A.Choudhary, "Techniques and Optimizations for Developing Irregular Out-of-Core Applications on Distributed-Memory Systems", Technical Report TR 96-4, Institute for Software technology and Parallel Systems, University of Vienna (1996).

[4] "MPI-2: Extensions to the Message Passing Interface", The Message Passing Interface Forum (1997). Available at `http://www.mpi-forum.org/`.

[5] P.Chen, G.Gibson, R.H.Katz and D.A.Patterson, "RAID: High Performance, Reliable Secondary Storage", ACM Computing Surveys, 26(2) p.145 (June 1994).

[6] P.F.Corbett *et al*, "Parallel File Systems for the IBM SP2 Computers", IBM Systems Journal, 34(2) p.222 (Jan 1995).

[7] A.Sweeney *et al*, "Scalability in the XFS File System", USENIX conference, San Diego, California (1996).

[8] "Paragon System User's Guide", Intel 312489-004 (May 1995).

[9] P.H.Carns, W.B.Ligon, R.B.Ross and R.Thakur, "PVFS: A Parallel File System for Linux Clusters", In Proc. of the Extreme Linux Track: 4th Linux Conf. (Oct 2000).

[10] C.H.Q.Ding, "Performance Characteristics of Parallel I/O on Cray T3E and Effective User Strategies", Available at: `http://www.nersc.gov/research/SCG/cding/io/paper.html`.

[11] "The Scalable I/O Initiative", Details available at: `http://www.cacr.caltech.edu/SIO`.

[12] "Scalable Performance Analysis: The Pablo Performance Analysis Environment", D.A.Reed *et al*, In Proceedings of the Scalable Parallel Libraries Conference, A.Skjellum (Ed.), IEEE Computer Society, p104, (1993).

[13] "The Grid: Blueprint for a New Computing Infrastructure", I.Foster and C.Kesselman (Editors), Morgan Kaufmann, San Fancisco, (1999).

[14] "Input/Output Characteristics of Scalable Parallel Applications", P.E.Crandall, R.A.Aydt, A.A.Chein and D.A.Read, Dept. of Computer Science, University of Illinois, Available at: `http://www-pablo.cs.uiuc.edu/Publications/Papers`.

[15] "Input/Output Characteristics of a Synthetic Aperture Radar Application", P.E.Crandall, A.A.Chein and D.A.Read, Dept. of Computer Science, University of Illinois, Available at: `http://www-pablo.cs.uiuc.edu/Publications/Papers`.

[16] "Data Sieving and Collective I/O in ROMIO", R.Thakur, WGropp and E.Lusk, Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation, p182 (Feb 1999).

[17] "NHT-1 I/O Benchmarks", R.Carter, B.Ciotti, S.Fineberg and B.Nitzberg, Report RND-92-016, NAS, NASA Ames Research Center (1992).

[18] "PPFS: A High Performance Portable Parallel File System", J.V.Huber, C.L.Elford, D.A.Reed, A.A.Chien and D.S.Blumenthal, In Proc. of the Intl. Conf. on Supercomputing (July 1995).

[19] "PASSION Runtime Library for Parallel I/O", R,Thakur, R.Bordawaekar, A.Choudhary, R.Ponnusamy and T.Singh, Scalable Parallel Libraries Conf. (Oct 1994).

[20] " Public International Benchmarks for Parallel Computers, PARKBENCH", R.W. Hockney and M. Berry (Editors), Committee: Report - 1 (Feb 1994).

[21] "Evaluating ParFiSys: A high-performance parallel and distributed file system", F.Perez, J.Carretero, F.Garcia, P.De Miguel and L.Alonso, Journal of Systems Architecture, vol. 43, no. 8, pp. 533-542 (1997).

[22] "High Performance Cache Management for Parallel File Systems", F. Garcia, J. Carretero, F. Pérez and P. de Miguel, 3rd International Meeting on Vector and Parallel Processing, p239-252 (June 1998).

[23] "HFS: A Flexible File System for Shared-Memory Multiprocessors" O.Krieger, PhD thesis, University of Toronto (Oct 1994).

[24] "Galley: A New Parallel File System for Scientific Applications", N.A.Nieuwejaar, PhD. thesis, Dept. of Computer Science, Dartmouth College, (Nov 1996).

[25] "VIPIOS: The Vienna Parallel Input/Output System", E.Schikuta, T.Fuerle and H.Wanek, in Proc. of the Euro-Par'98, Springer Verlag LNCS, Southampton, England (Sept 1998).

[26] "DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems", M.Beynon, T.Kurc, A.Sussman and J.Saltz, University of Maryland Technical Report CS-TR-4104 and UMIACS-TR-2000-04 (Feb 2000).

[27] "Implementing Journaling in a Linux Shared Disk File System", K.W.Preslan *et al*, To appear in Procs. of the 8th NASA Goddard Conf. on Mass Storage Systems and Technology (2000).

[28] "ChemIO: High Performance Parallel I/O for Computational Chemistry Applications", J.Nieplochat, I.Foster and R.A.Kendall, To appear in Intl. J. Supercomp. Apps. High perf. Comp. (1998).

[29] "Server-directed collective I/O in Panda", K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. In Proceedings of Supercomputing '95, San Diego, CA, IEEE Computer Society Press (1995).

[30] "The Scotch Parallel Storage Systems", G.A.Gibson, D.Stodolsky, F.W.Chang, W.V.Courtright II, C.G.Demetriou, E.Ginting, M.Holland, Q.Ma, L.Neal, R.H.Patterson, J.Su, R.Youssef and J.Zelenka, To appear in: Proceedings of the IEEE CompCon Conference, San Francisco (March 1995).

[31] "The High Performance Storage System", R.A.Coyne, H.Hulen, R.W.Watson, Proc. Supercomputing 93, Portland, OR, IEEE Computer Society Press (Nov 1993).

[32] "Experimental Evaluation of the Hewlett-Packard Exemplar File System", R.Bordawekar, S.Landherr, D.Capps and M.Davis, CACR Technical Report 143, California Institute of Technology (Sept 1997).

[33] "Using MPI-2", W.Gropp, E.Lusk and R.Thakur, The MIT Press (Jan 2000).