ORNL/TM-13440

Center for Computational Sciences

A STUDY OF I/O IN A PARALLEL FINITE ELEMENT GROUNDWATER TRANSPORT CODE

David Mackay [†]
Ed D'Azevedo [⋆]
Kumar Mahinthakumar [‡]

[†] Intel Corp. CO1-02,
15201 NW Greenbrier Pkwy.
Beaverton, OR 97006.

[⋆] Computer Science and Mathematics Division,
Bldg 6012, MS 6367,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831.

[‡] Center for Computational Sciences,
Bldg 4500N, MS 6203,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831.

Date Published: May 1997

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Lockheed Martin Energy Research Corp.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-96OR22464

Contents

List of Tables

List of Figures

Acknowledgements

We acknowledge the support by the Center of Computational Sciences at Oak Ridge National Laboratory, the Mathematics and Computer Science Division at the Argonne National Laboratory and the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign for the use of their supercomputing facilities.

A STUDY OF I/O IN A PARALLEL FINITE ELEMENT GROUNDWATER TRANSPORT CODE

David Mackay
Ed D'Azevedo
Kumar Mahinthakumar

Abstract

A parallel finite element groundwater transport code is used to compare three different strategies for performing parallel I/O: (1) have a single processor collect data and perform sequential I/O in large blocks, (2) use variations of vendor specific I/O extensions (3) use the EDONIO I/O library. Each processor performs many writes of one to four kilobytes to reorganize local data in a global shared file. Our findings suggest having a single processor collect data and perform large block contiguous operations may be quite efficient and portable for up to 32 processor configurations. This approach does not scale well for larger number of processors since the single processor becomes a bottleneck for gathering data. The effective application I/O rate observed, which includes times for opening and closing files, is only a fraction of the peak I/O performance. Some form of rearrangement and buffering of data in remote memory as performed in EDONIO may yield significant improvements for random I/O access patterns and short requests. Implementors of parallel I/O systems may consider some form of buffering as performed in EDONIO to speed up such I/O requirements.

1. Introduction

Efficient handling of multi-gigabytes of disk I/O in a parallel computing environment is a challenging problem for application programmers. The programmer is faced with a dilemma of using portable but sequential standard I/O constructs, or non-portable vendor extensions to take advantage of parallel I/O subsystems to achieve good performance.

This paper uses a parallel finite element groundwater transport code to explore and compare different strategies for performing parallel I/O. This application performs many writes of only several kilobytes in length. The small write sizes do not lend themselves easily to high performance I/O. In this paper we compare various parallel data writing strategies to determine the best I/O strategy for parallel programmers. These results should be of interest both to the parallel programmer and the high performance system designers. Our findings suggest having one processor gather the distributed parallel data and perform large block contiguous operations may be quite efficient and portable for small configurations of processors. Some form of rearrangement and buffering of data in remote memory as performed in the EDONIO I/O library may yield significant improvements for random I/O access patterns and short requests.

Several groups are performing active research in high performance portable parallel I/O solutions under the multi-agency Scalable I/O Initiative [9]. For example, the PABLO [14] performance analysis environment and PPFS (Portable Parallel File System) [16] are developed at University of Illinois. The PASSION (Parallel And Scalable Software for I/O) [15] compiler and run-time environment to optimize out-of-core applications written in HPF (High Performance Fortran) is being developed at the Center for Advanced Technology in Computer Applications and Software Engineering at Syracuse University. ADIO (Abstract-Device Interface for I/O) [4] and ROMIO [8] are efforts from the Scalable I/O Project at the Argonne National Laboratory.

The MPI Forum [5] is working on producing the MPI-2 standard that addresses portable and efficient parallel I/O. MPI-IO implementations are currently being developed and tested on the IBM SP2, Intel Paragon, Cray T3D, Meiko CS-2, and SGI Clusters by groups at Lawrence Livermore National Laboratories, and NASA Ames Research Center. When the MPI-IO standard is widely adopted by vendors with efficient implementations, MPI-IO might become the ideal solution for new applications. Until that time, alternative practical approaches are needed to solve today's needs.

To explore and evaluate different strategies for performing parallel I/O, we consider in detail a distributed memory parallel groundwater transport application based on structured grids on the Intel Paragon system and other parallel systems. We consider several approaches: (1) collection to one processor and perform sequential I/O, (2) using two variations of vendor supplied extensions, (3) using EDONIO, a software library to perform disk caching in distributed memory. Our timings on the Paragon supercomputer indicated that writes are generally more expensive than reads and hence we decided to focus only on writes in this paper.

The groundwater transport application, its nature and I/O needs, is described in §2 and §3. The Paragon system parallel file system is described in §4. Section 5 discusses the three strategies used and their performance on the Intel Paragon system. Some performance results on other parallel systems are included in §6. Finally our conclusions and findings are summarized in §7.

2. Groundwater Transport Application

The I/O application chosen here is from the finite-element transport module of the parallel groundwater remediation codes PGREM3D [11]. Below we briefly describe the principal governing equation, numerical implementation, and the parallelization strategy.

2.1. Governing Equation

The general governing equation describing the transport of a dilute solute in a saturated, essentially incompressible porous medium is [7, 10, 13]

$$\frac{\partial c}{\partial t} = \nabla \cdot (\mathbf{D} \nabla c) - \mathbf{v} \nabla c + G. \tag{1}$$

where c here represents the mass concentration of the solute in the dissolved phase $[M/L^3]$, t is the time $[T]$, $\mathbf{v}$ is the 3×1 velocity field vector $[L/T]$, $\mathbf{D}$ is a 3×3 dispersion tensor dependent on $\mathbf{v}$ $[L^2/T]$, and G is the source/sink term representing the rate of mass production or consumption $[M/L^3T]$. The term G can take many forms depending on the type of reaction or the source/sink term. In PGREM3D, G represents terms coming from injection wells, first order decay, kinetic/equilibrium sorption, and bioremediation. For the case of bioremediation, an additional transport equation for biomass of the form given by Equation (1) needs to be solved. The additional equations describing sorption kinetics and microbial growth/decay involve first order derivatives in time and zeroth order derivatives in space.

The velocity field $\mathbf{v}$ is usually obtained from the solution of the groundwater flow equation [12]. For the steady state, saturated flow $\mathbf{v}$ is given by

$$\theta \mathbf{v} = -\mathbf{K} \nabla h \tag{2}$$

where h is the computed pressure head field from the groundwater flow equation, $\mathbf{K}$ is the

3×3 hydraulic conductivity tensor (usually diagonal), and θ is the porosity. The elements of the 3×3 dispersion tensor $\mathbf{D}$ are given by

$$\mathbf{D}_{ij} = \alpha_L \|\mathbf{v}\| \delta_{ij} + (\alpha_L - \alpha_T) \frac{\mathbf{v}_i \mathbf{v}_j}{\|\mathbf{v}\|} + D_m \tag{3}$$

where α_L and α_T are longitudinal and transverse dispersivities assumed to be constant, D_m is the coefficient of molecular diffusion assumed to be constant (usually very small).

2.2. Numerical Implementation

In PGREM3D, the three-dimensional form of the transport equation is discretized using linear hexahedral elements on a logically rectangular grid and based on the upstream weighted Galerkin formulation [7]. The time stepping is implemented using a variable weighted finite-difference scheme. Options are available to handle uniform rectangular, non-uniform rectangular, and distorted grids. Different boundary condition options including time dependent and cyclic boundary conditions are available. The code executes a comprehensive mass balance check at each time step as outlined by [6]. The mass matrix and the zeroth order terms are evaluated using a lumped formulation [7]. Reactions include equilibrium sorption, kinetic sorption, first order decay and non-linear bioremediation. The entire matrix is assembled only during the first time step or when the boundary conditions change. The right hand side is assembled at all time steps. Non-linearity in bioremediation reactions are handled using Picard type iterations. Each of these iterations involve a linear system (matrix) solution. The linear system solution is performed using iterative solvers based on Krylov subspace methods.

The finite-element approximation of Equation (1) results in a matrix equation of the form $\mathbf{A}x = b$, where $\mathbf{A}$ is a sparse, non-symmetric matrix. In PGREM3D, the nodes and elements are numbered in the z-first ‘natural order’. For a logically rectangular grid structure and ‘natural ordering’ of unknowns matrix $\mathbf{A}$ has a 27-diagonal banded non-zero structure. In this implementation the non-zero entries of the matrix are stored by diagonals. This enables vectorizing compilers to generate extremely efficient code for operations like a matrix vector product, which are used in iterative Krylov solvers. Although PGREM3D has a choice of several Krylov solvers [13], the BiCGSTAB solver was used throughout our tests.

2.3. Test Problem

The test problem involves a single extraction well in the center of a square domain extracting a uniformly distributed cylindrical plume. This simple problem was chosen since the radial velocity field for this problem is analytically known and therefore eliminates the need for a flow solution. In this paper we are concerned with the I/O performance, rather than the complexity of the problem. The velocity field for the test problem is analytically obtained from the simple expression $\|\mathbf{v}\| = Q_w/(2\pi r d)$. Here Q_w is the pumping rate, r is the radius from the center of the well and d is the constant vertical depth. This solution assumes infinite boundaries. The pumping rate Q_w and the initial radius of the cylindrical plume r_0 are set to arbitrary constant values.

2.4. Parallelization

Our parallel implementation (in double precision) was originally targeted for the Intel Paragon machines using asynchronous `NX` message passing. The code was then ported to the SGI / Power Challenge Array, SGI / Cray Origin 2000, Convex Exemplar, and IBM SP systems using an MPI (Message Passing Interface) implementation.

For parallelization, we used a two-dimensional (2-D) domain decomposition in the x and y directions as depicted in Figure 1. A 2-D decomposition is generally adequate for groundwater problems because common groundwater aquifer geometries involve a vertical dimension that is much shorter than the other two dimensions. For the finite-element discretization such decomposition involves communication with at most 8 neighboring processors. We note here that a 3-D decomposition in this case would require communication with up to 26 neighboring processors.

To avoid additional communication during the assembly step we overlap a set of finite elements along processor boundaries. There is no overlap in node points. In order to preserve the 27-diagonal band structure within each processor submatrix, we perform a local renumbering of the nodes within each processor subdomain. This local renumbering causes numbers in only the z-y plane to have contiguous global ordering. Such a numbering gives rise to some complications during explicit communication and parallel I / O stages. For example, in explicit message passing, non-contiguous array segments had to be gathered into temporary buffers prior to sending. This buffering requires only one message for each neighboring processor. These are then unpacked by the receiving processor. In contrast the I / O must read or write globally contiguous data and can not pack or buffer output in the same fashion.

3. I/O Test Case, Needs and Requirements

3.1. I/O Test Case and I/O Needs

The solute transport module of PGREM3D requires parallel binary I / O for reading the velocity and nodal flux fields (output from the flow module) and for writing out the concentration fields at desired timesteps. A very small number of global input parameters are read from an ASCII file using one processor and then broadcast to all other processors. Additional I / O

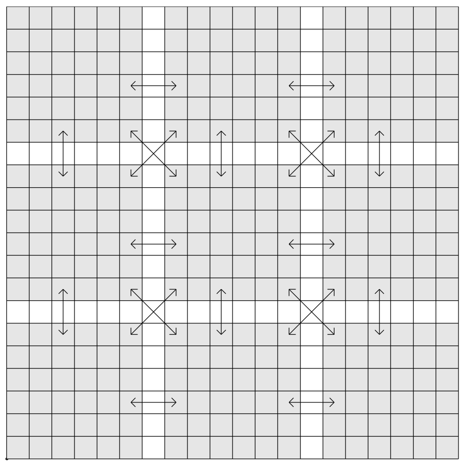

Figure 1: Plan View of Two-Dimensional Domain Decomposition. Each gray region belongs to a processor; the white regions are overlapped. The arrows show the communication pattern.

is required for reading the element model data and also in check-pointing for writing out and reading in binary restart files. When distorted meshes or fully heterogeneous material properties are encountered additional binary I/O is required to read in these values. All binary I/O is performed in the 'natural' z-first then y, then x order.

Parallel I/O to a single globally shared file requires non-contiguous writes to a file on small blocks of size in the order of a few KBytes. As we show later in this paper, this can contribute to some I/O performance degradation particularly when a large number of processors are involved. As described in §2.4, the 2-D domain decomposition strategy employed here results in a local processor node numbering that is different from the global node numbering of the entire domain. However, when the solution output is written to a file, we need to make sure that the proper order is preserved in the global sense.

For a typical finite-element application, some I/O quantities can be node based and some can be element based. For example, in PGREM3D, the quantities such as velocities and heterogeneous material properties (e.g. porosity, reaction coefficients etc.) are element based, and quantities such as concentrations, nodal fluxes, mesh coordinates are node based. Observe Figure 1 to note that although elements overlap between processors, there is no overlap in node assignment. Each processor is responsible for performing I/O on a unique set of elements and nodes, so that asynchronous independent parallel I/O can be exploited, even though there may be overlap of elements along processor boundary. Explicit message passing is then performed at the end of a read to get information at the overlapping elements where information is required but has not been read in.

For a typical simulation the binary writes are of the order nb bytes (where $nb = 8 * nx * ny *$

nz and *nx*, *ny*, *nz* are the problem dimensions in each direction) per output or restart frequency (hereby denoted as instance). The output and restart frequencies are controlled by the user. For a typical simulation, the output frequency can be as frequent as once every 10 timesteps and the restart frequency once every 1000 timesteps of simulation. For the simplest case of single component transport, the write size is *nb* bytes per instance, and for bioremediation involving kinetic sorption it is $5nb$ bytes per instance. Moreover, for each field variable (*nb* bytes), a file is opened and closed. For example, for a $251 \times 251 \times 11$ problem the output can be from $6to28$ MBytes per instance. If we use 16 processors for this simulation using a 4×4 domain decomposition (see Figure 1), then each contiguous block of write is approximately only 1.4 Kbyes ($= 8 \times 11 \times 251/4$).

3.2. File Requirements

In designing the parallel I/O routines for this code we decided to follow certain constraints. The first constraint is that each field variable would be stored in a single file in the global finite element node ordering. This means data from each processor would not be contiguous in the file (c.f. §3.1). Data from each processing node will be collected in global ordering into a single file. Data will not be stored based on local processor data order. There are some reasons we feel this is important. The output files are used both for postprocessing a history of solute movement and also for restart/checkpoint of the calculation. The scientist may postprocess the data on a different number of processors than used for calculations. When restarting the run to continue the calculations, there may be more or fewer processors available. If data is stored in individual files or in local processor ordering, it is more difficult to postprocess or restart calculations on a different number of processors. A single file makes it easier for the user to manipulate the data files, since there is only one file to copy or move to a long term storage device for backup and recovery. In addition, the files must be written to a file system that is always accessible. The data files should not be written to a file system belonging to a dedicated processor. Otherwise, the application can be restarted only on the identical processor configuration.

In addition, we choose to keep each instance of field variables in separate files. We can save on the overhead associated with opening and closing files, if we write all of the data to one large file. This means we will incur the cost of opening and closing each file each time we write data. However, it is common practice to write out the data for each field in a separate file. In postprocessing, the scientist may want to examine only one part of the data, say the soluble nutrient. Although this could be extracted from a large file, we followed PGREM3D code tradition of putting each variable in a separate file.

4. Paragon System Parallel I/O

The Intel Paragon supercomputer is a massively parallel computer built from processing elements using the 50 MHz Intel i860XP microprocessor and distributed memory interconnected in a two dimensional mesh. The system runs a version of OSF1 operating system. The system dedicates a number of processors to handle I/O tasks in a dedicated I/O partition, separate from the compute partition where applications are run. Each processor board in the I/O partition has a daughter card and RAID (Redundant Array of Independent Disks) attached to it. All output on the Paragon system is written to a RAID instead of a single disk. Each RAID has a UNIX file system (ufs) partition. On top of the UNIX file system, Intel built a parallel file system (pfs) which stripes a file across multiple RAID's and I/O processors. To the Paragon system user all of the standard UNIX file commands (ls, cp, rm, ftp) work on the parallel file system, and the striping is transparent.

The system administrator controls the maximum number of ufs partitions a parallel file system will stripe across and the size of the stripes. A user may call `fcntl` to stripe across fewer RAID's or to select a smaller stripe size, but he may not exceed the number of RAIDs or stripe size setup by the system administrator. As part of the `NX` message passing library, Intel provides optimal `cread` and `cwrite` commands to improve I/O performance. The goals in designing the parallel file system were to optimize I/O for large files, to build on top of standard UNIX file systems, and to allow normal UNIX file manipulation.

The Intel XPS/150 Paragon system at the Oak Ridge National Laboratory (ORNL) allows us to test I/O in a number of different manners. The ORNL XPS/150 Paragon system has four different parallel file systems on it, each striping across a different number of RAIDS (/pfs64, /pfs32, /pfs16, /pfs8), to meet the diverse needs of the different applications and users on the system. Although not part of its normal operating environment, we were able to write to a specially configured parallel file system on a single RAID to use for comparisons to other systems. Each compute node has 64MB of memory.

Each RAID can write at about 3.0 MBytes/sec. We measured writing files at 190 MBytes/sec across 64 I/O nodes (2.97MBytes/sec per processor) on the XPS/150, by excluding the file open and close times. However, opening a single file striped across 64 RAIDS each connected to a different processor on the network and establishing ports to 1024 compute processor takes from 10 to 30 seconds depending on the I/O mode. This is a significant amount of time, especially for small files. In all timings shown in this paper we measure application output time, this includes the time to open, write and close the file. This is not the actual device write rate, but more importantly though, this is the time cost a scientist running an application observes.

5. Strategies for Performing Parallel I/O

		Write rate in MB/sec						
File Size	PFS stripes	1024 nodes	512 nodes	256 nodes	128 nodes	64 nodes	32 nodes	16 nodes
2 GB	64	51	77	113	122	132	136	99
8 GB	64	104	125	144	151	152	155	107
2 GB	32	44	50	62	68	74	78	
8 GB	32	66	125	75	77	76	82	
2 GB	16	29	31	35	37	38	38	
8 GB	16	36	36	37	38	38	38	

Table 1: Application write rates on Intel Paragon system in MB/sec

The results shown in Table 1 for the Intel Paragon system for writing files, are an ideal case. Each node was writing the same amount of data, and the amount of data written at each call exactly matched the parallel file system stripe size. This is not the case for typical applications. Even though the code studied in this paper operates on a structured grid, the number of finite elements and/or nodes assigned to each processor may differ in each direction. As previously described, the finite element groundwater transport code solves a time dependent problem. As part of the output, a history file is needed that shows the movement and tracing of particles as they are transported over time is needed. The code will generate several history files every hour. Since the application operates on a logical three-dimensional cube, it is easier to plan the output than in an arbitrary unstructured case. In an unstructured finite element grid, the ordering for global data stored in output files may differ dramatically from the ordering for local data on a processor. As a result, the file access from each node may be very irregular and very difficult to optimize. In other words, the structured I/O case is easier to optimize than the unstructured case.

In the simplest groundwater transport case, there may only be one solute whose transport is simulated. When this code is used for bioremediation with kinetic sorption though, at least five different field values must be tracked: a dissolved contaminant, a solid phase contaminant, a dissolved phase nutrient, a solid nutrient, and biomass. In some cases there may be multiple contaminant or nutrients. Each of these field values is stored in an array of the same length. Each of these variables is written in a separate file, and each file is the same size. In our tests we measure the time to write one file and calculate the write rate. In a typical one hour run of the code, we would actually want to write a history file for each property four or five times per hour. So the actual I/O time would be 20 times what we measure here. Since all files are the same length and are written to the same directory, we feel it is fair to measure time to write one file and extrapolate.

We implemented four different methods of performing the output on the Intel Paragon supercomputer. The first method of writing the output file was to have each processor send its data to the first processor. The first processor opened the file and wrote out all data. The first processor received, stored, and wrote the data using a 128Kbyte buffer. Flow control was added to handle the flood of incoming messages from the other processors. This method is not ideal, but is portable to a wide variety of systems running different message passing environments, such as `NX, MPI, PVM`. We implemented this method in order to compare with other systems.

The second case followed the standard Intel Paragon system parallel I/O model. That is the file was opened for asynchronous I/O, and each processor repositioned its own file pointer to the appropriate file location and wrote its output. Because of the distributed nature of the problem, each processor called `lseek()` multiple times to fill the file. The third method used is a variation of the second method. The only change is the addition of one system call to buffer data on the I/O processors. The application makes the same number of calls to `lseek()` and `cwrite()`.

The fourth method used a library package called EDONIO described in §5.1. This simple change essentially required only changing `lseek/cwrite` commands to `dolseek/dowrite` commands and linking in the appropriate library. We attempted a fifth method where we made another system call to match the file stripe parameter to the application write lengths. This method showed no improvement and is not reported here.

5.1. Distributed Object Network I/O Library

EDONIO (Distributed Object Network I/O Library) [1, 2, 3] provides fast parallel direct access random I/O operation to a global shared file by providing a large multi-gigabyte disk cache using the aggregate distributed memory. EDONIO translates I/O requests to message communication. On most modern multiprocessors, such as the Intel Paragon system, the communication network bandwidth is commonly much higher than the I/O subsystem. Thus it may be faster to transfer data to/from memory on remote processors than directly from the disk. EDONIO also reorganizes actual disk operations to perform transfers in large contiguous blocks aligned to RAID disk geometry. User initiated prefetching and cooperative writing to enhance I/O performance are also supported in EDONIO.

EDONIO mimics the UNIX like capability in `lseek/read/write` for global shared files. This facilitates porting of serial code or conversion of `NX` codes to EDONIO. To access a shared file, each processor uses `lseek` to relocate its own private copy of the file pointer and then performs input/output operations. Simultaneous output to overlapping regions in a shared file is non-deterministic, which is similar to ‘reckless’ mode in other parallel file systems. Unlike MPI-IO, EDONIO does *not* provide high level description of global I/O operations such as writing out a two-dimensional distributed block cyclic array. Instead, EDONIO provides simple yet flexible primitives and still manages to achieve high performance.

EDONIO organizes the disk file in fixed size blocks distributed in a block cyclic fashion across processors. The block size is commonly chosen to match the RAID stripe size (64KBytes on the Intel Paragon). On the Intel Paragon, each processor uses the fastest `NX` specific `M_ASYNC` file access mode to perform independent I/O requests. A simple “Least Recently Used” algorithm is used on each processor to cache data blocks in a dedicated pool of memory. The cache size is user adjustable (EDONIO uses a maximum of 4MBytes per processor by default). A read reference to a missing block will cause the data block to be transferred from disk, similarly a write reference may cause overflow data blocks to be drained out to the disk. EDONIO performs a cooperative filling or purging of disk cache on a user initiated ‘prefetch’ or ‘flush’ operation. In all of the results in this paper we include the EDONIO flush command as part of the write time. The EDONIO Library package must still open the file from every node, write the data, and close the file. The advantage is that the data reorganization is hidden from the programmer, and, in addition the data reordering provides large contiguous data blocks to write which improves I/O device performance.

5.2. Results on Paragon system

We present the results on the Paragon system here. Write rates for other platforms are presented in the next section. Table 2 shows the finite element grid size and the output file sizes for each run. Tables 3 and 4 show complete results for all four strategies and different pfs stripe configuration. In Figures 2 and 3 we have plotted the data for /pfs64 and /pfs8. We expected gathering data to one node would perform poorly when writing from hundreds of processors. This method actually performed fairly well up to 32 processors and sometimes up to 64 processors. The standard `NX` write routines worked fairly well. By examining the write rates for 64 processors in Table 4 shows, we see that buffering on the I/O processors can significantly improve write performance. On the other hand in Figure 3 we notice that write performance degrades when the system attempts to buffer a file from 1024 compute processors when there is insufficient memory on the /pfs8 file system. This is not surprising, and in addition the ratio of 1024 compute processors to 8 I/O processors is outside the recommended ratio for optimal pfs performance. The EDONIO library package consistently showed the best write performance. We believe this was due to a better buffer data collection process. Collectively these results indicate that a significant amount of time in performing output is spent gathering small segments of data into the proper global order.

processors	grid size	file size
1024	$1801 \times 2001 \times 11$	302.4 MB
512	$1401 \times 1262 \times 11$	148.4 MB
256	$901 \times 1001 \times 11$	75.7 MB
128	$701 \times 631 \times 11$	37.1 MB
64	$451 \times 501 \times 11$	19.0 MB
32	$251 \times 251 \times 11$	5.3 MB

Table 2: Problem size and file size.

	/pfs64				/pfs32			
processors	one	std	buf	donio	one	std	buf	donio
1024	2.1	5.4	5.6	10.3	2.2	4.0	4.0	11.5
512	2.5	6.3	6.6	9.1	2.6	4.4	5.2	10.6
256	2.6	5.2	6.0	9.0	2.6	3.9	5.4	10.1
128	2.3	2.9	5.9	6.6	2.5	3.5	5.7	6.7
64	2.2	2.2	3.8	5.3	2.3	3.3	4.8	6.8
32	1.8	0.9	1.2	1.7	1.6	0.7	2.0	2.7

Table 3: Effective write rate in MBytes/second on Intel Paragon system

	/pfs16				/pfs8			
processors	one	std	buf	donio	one	std	buf	donio
1024	2.2	2.4	1.8	10.3	2.2	1.2	0.8	7.6
512	2.4	2.7	2.6	9.6	2.7	1.4	1.1	8.4
256	2.5	2.2	2.9	11.0	2.5	1.2	1.3	7.3
128	2.8	2.3	3.5	8.0	2.7	1.3	1.9	7.4
64	2.5	1.9	4.2	7.0	2.3	1.1	2.5	6.4
32	2.2	0.6	1.8	2.6	2.4	0.6	1.8	3.0

Table 4: Effective write rate in MBytes/second on Intel Paragon system

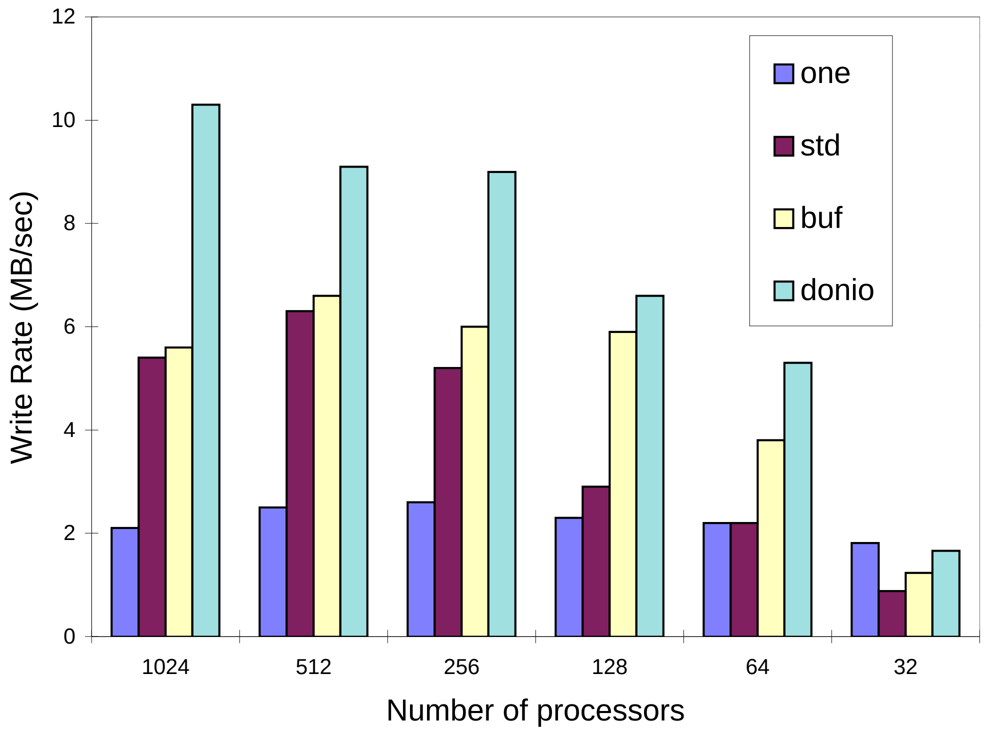

Figure 2: Write rates for PGREM3D to /pfs64 on Intel Paragon system

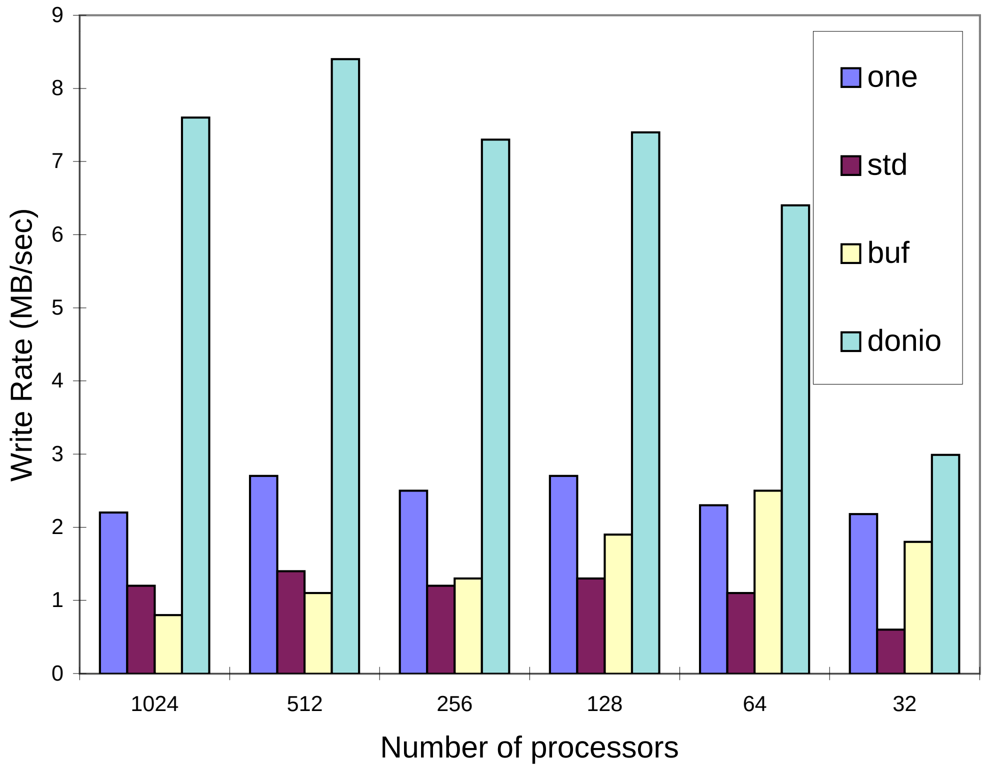

Figure 3: Write rates for PGREM3D to /pfs8 on Intel Paragon system

6. Results on Multiple Platforms

Although our performance evaluations were primarily targeted towards the Intel Paragon, we measured and compared performance across a range of distributed parallel platforms for a test problem involving moderate number of processors (8, 16, and 64). The comparisons involve SGI / Power Challenge Array (16 processors), Convex Exemplar (8 processors), Cray / SGI Origin 2000 (16 processors), Intel Paragon XPS / 150 (16 and 64 processors), and IBM SP (16 and 64 processors). Only the single-node I / O mode was used in this comparison since not all systems have true parallel I / O capabilities. The problem size is chosen to allocate approximately 43,000 nodes / processor. This translates to $251 \times 125 \times 11, 251 \times 251 \times 11$, and $501 \times 501 \times 11$ for the 8, 16, and 64 processor configurations.

Below in Table 5 we compare the I / O rate for the 16 processor case (except Convex Exemplar where we used 8 processors). We note here that the I / O rates include times for message

System	MB / s
SGI / PC(ufs)	0.49
Convex Exemplar(scratch)*	3.71
Convex Exemplar(ufs)*	1.58
Cray / Origin 2000(ufs)	1.88
IBM SP(ufs)	1.46
IBM SP(/ piofs)	8.08
XPS / 150(/ pfs_1)	1.29
XPS / 150(/ pfs64)	2.13

*8 processors with $251 \times 125 \times 11$

Table 5: Write rates for a $251 \times 251 \times 11$ problem size with 16 processors

passing required to collect data into one processor and times for opening and closing files. Hence the message passing latency and bandwidth of each system may impact the rates reported here. We also note that for some of the results presented here the files were written to the standard UNIX file system (ufs) of the user's home area. In Table 6 we present the write rates for the $501 \times 501 \times 11$ problem using 64 processors in the IBM SP and XPS / 150. It is interesting to note that the IBM SP performance on / piofs has decreased from the 16 processor case while the XPS / 150 performance on / pfs64 has increased slightly.

System	MB / s
IBM SP(ufs)	1.45
IBM SP(/ piofs)	6.62
XPS / 150(/ pfs_1)	1.35
XPS / 150(/ pfs64)	2.38

Table 6: Write rates for a $501 \times 501 \times 11$ problem size with 64 processors

In Table 7 we compare the total run time and write times for a typical run. This analysis gives us a measure of relative time spent on parallel I/O compared to the total time for each system. Unlike the Paragon system runs, these problems involved kinetic sorption and bioremediation where we required output of 5 vectors each of size $nx * ny * nz$. The solution requires nonlinear iterations with about 10 matrix solves per time step. The I/O performance analysis is for writing binary output at the end of 10 time steps. Each vector was written to a different file. The timings on 16 processor are for the $251 \times 251 \times 11$ problem except the Convex Exemplar which solved a $251 \times 125 \times 11$ problem using 8 processors. From Table 7 we observe that for Convex and IBM SP the time spent in binary output is less than 4% of to the total time.

System	Write Time	Total Time
SGI/PC(ufs)	54.3	435
Convex Exemplar(scratch)*	3.5	860
Cray/Origin 2000(ufs)	14.0	196
IBM SP(/piofs)	3.3	131
XPS/150(/pfs64)	12.4	354

*8 processors with $251 \times 125 \times 11$

Table 7: Total and output times in seconds for a $251 \times 251 \times 11$ problem size with 16 processors

7. Discussion

We did not expect exceptionally high write rates for the chosen application. Due to the global 'natural' z-fastest node ordering, each processor writes out a contiguous set of data values that correspond to only a plane of the mesh. For the problem sizes considered, each request is only 2 to 4KBytes in size. We believe it may be difficult to obtain high throughput with small write segments when the I/O system is optimized for high volume access to large files.

The all-to-one method worked fairly well up to 32 processing nodes by coalescing multiple short write requests into a larger 128KBytes block. However, this strategy does not scale well as the single I/O processor becomes a bottleneck collecting the data. Moreover, a single processor cannot saturate the bandwidth to a highly parallel /pfs32 or /pfs64 file system.

The standard access mode using `NX` specific `gopen/lseek/cwrite` has better scalability for over 128 processors writing to /pfs64 or /pfs32. Using the buffered I/O on the dedicated I/O nodes of the Paragon has marginal improvement on performance. The limited amount of memory (32Mbytes per node) on the dedicated I/O nodes may not be sufficient to have a significant impact. On /pfs8 and 1024 processors, buffering actually decreased performance.

The EDONIO library consistently gave the best performance. Since EDONIO utilizes distributed memory across all processors for data buffering, the entire file can in fact be easily cached in memory. Data is flushed out to disk in large blocks in the most efficient manner.

Data buffering is not a new idea. Early high performance vector systems used solid state memory storage devices as a disk cache before writing to disk. This type of buffering for reorganizing and sorting the data would benefit I/O on unstructured meshes in finite element applications.

We examined the MPI-IO specifications recommended by the MPI-2 forum. We did not have an implementation of MPI-2 to test this method on, but we observed two items that deserve comment. The MPI-2 specifications permit us to describe how data from each processing node is distributed into a global file. Then the data may be written with a single high level MPI-IO call which writes out the appropriate array. It seems to take just as much work for the programmer to describe the data layout to MPI as it does to calculate the offsets and call lseek between each write to the file. If there is a substantial overhead cost associated with generating the optimal schedule for this high level global write operation, then a reasonable MPI-IO implementation should amortize this cost even when multiple files are opened and closed.

For I/O on unstructured grids, a simple global description of the data distribution via `MPI_TYPE_CREATE_DARRAY` may not be convenient or feasible. The programmer may need to resort to generating short requests with various offsets. There will always be some applications, like PGREM3D, writing small pieces of data into very large files. In these cases it is our experience that the handling of the data-gather is just as important as the actual device throughput. The I/O buffering provided in the Paragon system pfs improved performance, but never matched the gathering and buffering strategy of EDONIO. Based on our results, the implementors of MPI-IO might consider an EDONIO like buffering scheme to redistribute and coalesce data into large blocks in preparation for efficient I/O to the parallel file system.

The purpose of this study was to examine parallel I/O from an application programmers point of view, in this case for a finite element program. We also feel that the high performance computing community should put as much emphasis on application I/O times as peak device I/O throughput. There was little variation in the gather-to-one processor write method used on all systems. The wide variations in parallel write rates observed by this code indicate that both programmers and system designer consider how to efficiently handle data reordering for I/O. In conclusion, we recommend that parallel applications writers consider an efficient I/O gather strategy, such as used in EDONIO.

8. References

[1] E. F. D'Azevedo and C. H. Romine. DOLIB: Distributed object library. In D. H. Bailey, P. E. Bjorstad, J. R. Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, and L. T. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 750–755, Philadelphia, 1995. SIAM.

[2] E. F. D'Azevedo and C. H. Romine. DONIO: Distributed object network I/O library. Technical Report ORNL/TM-12743, Oak Ridge National Laboratory, 1995.

[3] E. F. D'Azevedo and C. H. Romine. EDONIO: Extended distributed object network I/O library. Technical Report ORNL/TM-12934, Oak Ridge National Laboratory, 1995.

[4] ADIO: Abstract-Device Interface for I/O. Online document and links at `http://www.mcs.anl.gov/home/thakur/adio/`.

[5] MPI Forum. The message passing interface (mpi) standard. Online documentation and links at `http://www.mcs.anl.gov/mpi/`.

[6] P. S. Huyakorn, J. W. Mercer, and D. S. Ward. Finite element and mass balance computational schemes for transport in variably saturated porous media. *Water Resources Research*, 21:346–358, 1985.

[7] P. S. Huyakorn and G. F. Pinder. *Computational Methods in Subsurface Flow*. Academic Press, New York, 1983.

[8] ROMIO: A High Performance Portable MPI-IO Implementation. Online document and links at `http://www.mcs.anl.gov/home/thakur/romio/`.

[9] The Scalable I/O Initiative. Online documentation and links at `http://www.cacr.caltech.edu/SIO/`.

[10] J. D. Istok. *Groundwater modeling by the finite element method*. Water Resources Monograph 13. American Geophysical Union, Washington, D.C., 1989.

[11] G. Mahinthakumar. PGREM3D: Methods and user's guide. In Preparation ORNL/TM-13435, Oak Ridge National Laboratory, 1997.

[12] G. Mahinthakumar and F. Saied. Distributed memory implementation of multigrid methods for groundwater flow problems with rough coefficients. In A. M. Tentner, editor, *Proceedings of the Simulation Multiconference: High Performance Computing 1996*, pages 51–57, San Diego, 1996. Simulation Councils, Inc.

[13] G. Mahinthakumar, F. Saied, and A. J. Valocchi. Comparison of some parallel Krylov solvers for large scale contaminant transport simulations. In A. M. Tentner, editor, *Proceedings of the Simulation Multiconference: High Performance Computing 1997*, pages 134–139, San Diego, 1997. Simulation Councils, Inc.

[14] The University of Illinois PABLO Research Group. Online documentation and links at `http://www-pablo.cs.uiuc.edu/`.

[15] Parallel and Scalable Software for I/O. Online documentation and links at `http://www.cat.syr.edu/passion.html`.

[16] Portable Parallel File System (PPFS). Online documentation and links at `http://www-pablo.cs.uiuc.edu/Projects/PPFS/ppfs.html`.

ORNL/TM-13440

INTERNAL DISTRIBUTION

1.	A. S. Bland	21.	C. E. Oliver
2.	T. S. Darland	22.	S. A. Raby
3–7.	E. F. D'Azevedo	23.	B. A. Riley
8.	J. P. Gwo	24.	R. F. Sincovec
9.	K. L. Kliewer	25.	Laboratory Records - RC
10.	M. R. Leuze	26–27.	Laboratory Records Department / OSTI
11–15.	D. R. Mackay	28.	Central Research Library
16–20.	G. Mahinthakumar		

EXTERNAL DISTRIBUTION

29. Daniel A. Hitchcock, ER-31, Acting Director, Mathematical, Information, and Computational Sciences Division, Office of Computational and Technology Research, Office of Energy Research, Department of Energy, Washington, DC 20585

30. Frederick A. Howes, ER-31, Mathematical, Information, and, Computational Sciences Division, Office of Computational and Technology Research, Office of Energy Research, Department of Energy, Washington, DC 20585

31. Tom Kitchens, ER-31, Mathematical, Information, and, Computational Sciences Division, Office of Computational and Technology Research, Office of Energy Research Department of Energy, Washington, DC 20585

32. David B. Nelson, ER-30, Associate Director, Office of Energy Research, Director, Office of Computational and Technology Research, Department of Energy, Washington, DC 20585