to appear in Int. J. Supercomp. Apps. High Perf. Comp. 1998
(special issue on I/O in parallel applications)

ChemIO: High-Performance Parallel I/O for Computational Chemistry Applications

Jarek Nieplocha†
j_nieplocha@pnl.gov

Ian Foster✞
foster@mcs.anl.gov

Rick A. Kendall†
ra_kendall@pnl.gov

†Pacific Northwest National Laboratory
Richland, Washington 99352-0999

✞Argonne National Laboratory
Argonne, IL 60439

http://www.mcs.anl.gov/chemio/

Abstract

Recent developments in I/O systems on scalable parallel computers have sparked renewed interest in out-of-core methods for computational chemistry. These methods can improve execution time significantly relative to "direct" methods, which perform many redundant computations. However, the widespread use of such out-of-core methods requires efficient and portable implementations of often complex I/O patterns. The ChemIO project has addressed this problem by defining an I/O interface that captures the I/O patterns found in important computational chemistry applications and by providing high-performance implementations of this interface on multiple platforms. This development not only broadens the user community for parallel I/O techniques but also provides new insights into the functionality required in general-purpose scalable I/O libraries and the techniques required to achieve high- performance I/O on scalable parallel computers.

1 Introduction

Computational chemistry refers to the use of computational techniques to explore the chemical and physical interactions of atoms and molecules based on the fundamentals of quantum mechanical theory [1]. The accuracy of these theories makes computational techniques extraordinarily useful (e.g., the accurate prediction of thermodynamic properties of chemical reactions), but at the same time the computational complexity means that a chemist's ability to solve interesting problems is typically limited by available computational capabilities. For this reason, computational chemists were early adopters of scalable parallel supercomputers and now make extensive use of this technology (c.f., [2,3,4]).

To date, most chemistry algorithms implemented on scalable parallel systems have avoided the use of secondary storage devices primarily because of poor I/O characteristics. As I/O systems mature, however, we see increased interest in out-of-core methods. In principle, these methods can allow the solution of larger problems or, in some cases, faster solutions to fixed-size problems. However, the complex I/O patterns found on scalable parallel computers make the development of efficient, portable out-of-core programs difficult.

We believe that high-level I/O libraries can hide some of the complexity of the underlying I/O patterns from the programmer. (This belief is motivated in part by our experience in the related area of interprocessor communication, where support provided by the Global Arrays library [5,6] for a high-level, shared-memory view of distributed arrays has significantly simplified application development.) Unfortunately, while a large number of such I/O libraries have been developed, no existing library meets the requirements of computational chemistry applications. We can address this problem in one of two ways. We can design an ideal "universal" parallel I/O library, or we can develop an I/O library that meets the distinct requirements of chemistry codes. We have chosen

the latter course in the ChemIO project, believing that this will allow us to make more rapid progress, while also contributing to an understanding of the functionality required in an ideal library.

The goal of the ChemIO project is to develop techniques for high-performance I/O specialized to the requirements of large-scale computational chemistry problems. In the past three years, we have surveyed the I/O requirements of chemistry applications and have defined a standard I/O application programmer interface (API) that meets these requirements. The resulting ChemIO API supports three distinct I/O models (see Figure 1):

1. *Disk Resident Arrays (DRA)*. This component extends the Global Array model to support explicit transfer between global memory and secondary storage, allowing the programmer to manage the movement of array data structures between local memory, remote memory, and disk storage. This component supports *collective* I/O operations, in which multiple processors cooperate in a read or write operation, hence enabling certain optimizations.

2. *Exclusive Access Files (EAF)*. This component supports independent I/O to and from scratch files maintained on a per-processor basis. It is used for out-of-core computations in codes that cannot easily be organized to perform collective I/O operations.

3. *Shared Files (SF)*. This component supports the concept of a scratch file shared by all processors. Each processor can perform noncollective read or write operations to an arbitrary location in the file. (The application must handle mutual exclusion.)

We have developed high-performance implementations of each of these three components on multiple high-performance computer systems and have worked with chemists to apply the ChemIO API to a variety of computational chemistry problems. These applications have allowed us to study the utility and performance of our implementations in practical settings.

In this paper, we report on the computational chemistry applications that motivated the design of the ChemIO system, our experiences developing ChemIO components, and some early results obtained with the system on the current MPP platforms. The rest of the paper is as follows. In Section 2, we provide a brief review of major computational chemistry methods from an I/O perspective. In Section 3, we summarize the I/O requirements and discuss the approach we adopted to meet these requirements. In Section 4, we provide a description of the ChemIO modules and the techniques used to implement them. In Section 5, we discuss experiences with the ChemIO libraries. In Section 6, we summarize our results and outline plans for future work.

2 Computational Chemistry and I/O

We first review three specific methods commonly used in computational chemistry applications, in order to motivate the discussions of I/O requirements in Section 3. We also describe requirements for checkpoint/restart capabilities. Most of these applications have an inherently dynamic, many-to-few mapping of computation (e.g., the integral values that are to be computed) to data (e.g., elements of a Fock matrix or CI Hamiltonian matrix). This dynamic mapping motivate the use of a shared-memory programming model such as Global Arrays. HPF or Fortran-90 with message passing could be utilized but the complexity of the algorithms (as described in the following sections) and resulting implementations would be much greater since neither Fortran-90/message-passing nor HPF provide efficient one-sided access to distributed data structures. Furthermore, in disk-resident algorithms, standard I/O data access patterns (e.g., producer/consumer, nearest neighbor etc.) do not apply. Evolving message passing and HPF standards may in the future provide the required functionality, but are not feasible in the short to medium-term time frame.

2.1 The Hartree-Fock Method

The Hartree-Fock (HF) or Self-Consistent Field (SCF) method is the most widely used ab initio electronic structure method in computational chemistry and the starting point for many of the more accurate correlated

methods such as second-order Möller-Plesset (MP2) or coupled cluster single and double excitations (CCSD) [1]. The HF algorithm is iterative, with the time-critical step at each iteration being the formation of the Fock matrix. The integrals over the atomic basis functions are contracted with the density matrix to generate the Fock matrix. The Fock matrix is then diagonalized to generate the orbitals and density matrix to be used in the next iteration. The iterative procedure is repeated until convergence in energy or density is achieved, which usually requires about a dozen cycles. Alternatively, second-order methods may be used that avoid the diagonalization step by computing the change in the orbitals directly [7,8]. This process still involves construction of the Fock matrix or Fock-like matrices.

The atomic integrals are one- and two-electron integrals over the atomic basis functions, which are generally atom-centered Gaussian functions. The number of one-electron integrals scales as N^2, where N is the number of atomic basis functions and is proportional to the number of electrons in the chemical system. The number of two-electron integrals scales as N^4 and is thus the largest cost factor [9]. Each two-electron integral contributes to potentially six Fock matrix elements. The traditional HF algorithm (i.e., that used on workstations and vector supercomputers) is an out-of-core method that computes the nonzero atomic integrals once, stores them on disk ($O(N^4)$ double-precision values), and then retrieves them as required at each iteration. This algorithm is also sometimes called a completely disk-resident algorithm. An alternative, so-called direct algorithm, is also possible (and widely used on parallel computers); this recomputes the required integrals at each iteration [9,10].

Proper screening of the integrals to be computed is key to both the direct and traditional algorithms [7,10]. Aggressively using the Schwarz inequality at both the atom and shell level can reduce the number of integrals that need to be computed or stored and retrieved [11]. In the HF algorithm, the integrals are contracted against the density matrix. If density matrix values are small, the overall contribution to the Fock matrix is also small, and a direct algorithm need not recompute all integrals. It is harder to avoid recomputation in the traditional algorithm because the density matrix changes every iteration. However, a similar technique can be used to reduce disk accesses, provided the files are properly sorted and the integrals are accessible in a random but deterministic manner (i.e., a specific group or block of integrals can be accessed on demand).

We see from this discussion that the relative benefits of the direct and traditional algorithms depend on the costs of computing an integral and retrieving it from disk. In the following we explain these costs for a single example, namely a 512-node Intel Paragon. While the Paragon is no longer a state-of-the-art parallel computer, it does allow us to illustrate the tradeoffs that can arise.

A 512-node Paragon can calculate chemical properties of systems with up to 3000 basis functions and has routinely been used to compute properties of systems with a few hundred basis functions. For a calculation with 350 basis functions and no symmetry, the total number of integrals that potentially must be stored is 1.88 billion ($350^4/8$; the permutational symmetry reduces the total number of two-electron integrals that need to be computed by a factor of eight), or 15 GB. If we assume the aggregate disk bandwidth on parallel computers of 30 MB/s and ignore sparsity, a Paragon implementation of the traditional algorithm would take over 8 minutes to read the integrals at each HF iteration. The average cost of an integral is 300 to 500 flops, depending upon the basis set used. On the Paragon, a HF code can achieve 100 MFlops/s/node in a well-tuned Fortran code. The parallelization of the integral computation is straightforward and nearly embarrassingly parallel [12]. Hence, we can compute the integrals (that took 8 minutes to read) in 156 minutes on 1 node, in 1.2 minutes on 128 nodes, and in 18 seconds on 512 nodes. We see that, whereas the traditional method may be superior for small numbers of nodes (but note that we require enough nodes to achieve the full disk bandwidth of the system), the direct algorithm is much more scalable.

Because the amount of computation scales faster with N than the number of integrals, the tradeoffs are somewhat different when dealing with larger problems. For example, with 3000 basis functions we have 10.1 trillion integrals (81 TB). Reading the integral data set once would require 93 hours at 30 MB/s, given enough disk capacity to store the data (not likely on today's machines!). A direct algorithm would require 109 hours on 128 nodes and 27.5 hours on 512 nodes.

While future parallel computers can be expected to have faster I/O systems (by virtue of greater parallelism and faster devices), trends show computer speeds increasing more rapidly than I/O device speeds for the foreseeable future. Hence, we might conclude that there is little hope for the traditional algorithm. However, an additional factor must be considered: The time required to compute an integral can vary widely, with some integrals requiring an order of magnitude or more time than others. Hence, researchers have combined the traditional and direct algorithms to yield a hybrid (also called semi-direct) algorithm [13]. This algorithm estimates the relative cost of groups of integrals, at the level of either the shell quartet or groups of shell quartets. Integrals that are both expensive and commonly used are stored on disk; all other integrals are computed as needed. This algorithm, coupled with various modifications to the direct algorithm to reduce the total number of integrals computed, shows the most promise for the use of HF theory on scalable parallel supercomputers. The hybrid algorithm reduces the aggregate size of the integral data set being read from disk.

In summary, implementations of the HF method can significantly improve performance on scalable parallel computers *if* they are able to obtain close to peak I/O performance for sophisticated I/O patterns. Experience shows that obtaining this level of performance is difficult. We require sophisticated I/O tools that can distribute integrals properly for computation.

2.2 Multi-Reference Configuration Interaction

The multi-reference configuration interaction (MRCI) method is widely used in electronic structure computational chemistry to obtain accurate predictions of properties of chemical systems. The details of the algorithms and methodology used in the COLUMBUS MRCI code are given elsewhere [14]; here we outline the necessary components to determine the I/O requirements (cf. [15]). The basic algorithm involves constructing a matrix-vector product, $H\sigma$, of the CI Hamiltonian, H, and the trial or guess vector, σ. Both $H\sigma$ and σ are segmented in a fashion to allow portions of the molecular integrals to be used to form the partial matrix-vector product using only the segments. This is the basis of the parallel task generation for the COLUMBUS MRCI code. For modest to large chemical systems with basis sets that give accurate results, the rank of the H matrix is 10^6 to 10^9, but less than 10 percent sparse. Since both the σ vector and the $H\sigma$ vectors are too large to maintain in a single computer's memory, in workstations and vector supercomputers MRCI codes are stored and retrieved from disk each iteration.

Initial parallel implementations of the MRCI method were modeled on vector supercomputer codes and hence used sequential, direct access files to store data on disk. This approach had significant performance problems on scalable parallel computers because it was unable to exploit high-performance parallel I/O systems. Dachsel et al. [14] approached this problem by developing an MRCI implementation that replaced accesses to sequential and direct access files with I/O mechanisms that feed the requisite data to distributed-memory resident files using the Global Arrays toolkit, replicated data storage if enough memory is available, or a mix of these two options. This approach exploits the large aggregate memory of typical scalable parallel computers to increase performance relative to out-of- core codes, but has the significant disadvantage of limiting the size of the chemical system that can be computed to the aggregate memory of the supercomputer.

A natural extension of the existing approach is to employ parallel I/O to extend the memory-resident files that store the $H\sigma$ and σ vectors to the disk. Since the computational tasks are independent but must access various segments of the two vectors, the I/O mechanism must support a random access pattern to these segments or have one- sided asynchronous read operations with atomic update or write operations. Such an algorithm could take advantage of the nonblocking I/O to overlap read/write operations with computations. The asynchronous I/O to the shared files allows a triple buffering operation, overlapping I/O and computation, input of segment pairs, computation of matrix- vector product segments, and output of the updated matrix vector product.

2.3 RI-SCF and RI-MP2

The RI-SCF algorithm is a modification of the SCF (or Hartree-Fock) algorithm outlined above. Similarly, the RI-MP2 algorithm is a modified MP2 algorithm; the details are presented in [16]. The RI-SCF and RI-MP2 algorithms use the resolution-of-the-identity (RI) approximation to reduce the overall computational cost of their respective methodologies by replacing the complete 4-index integrals with a sum of a smaller set of 3-index and/or 2-index integrals that are cheaper to compute [16,17,18] and therefore reduce the overall I/O that must be done. This process involves the traditional atomic orbital (AO) basis set and a fitting basis set. The two steps of both models that involve I/O to secondary storage are the integral generation (output) and the formation of the RI-based Fock matrix or the RI- MP2 energy (input).

The mathematical transformation of the RI approximation is

$$\langle ij|kl\rangle = \sum_{tu} \langle ij|t\rangle V_{tu}^{-1}(u,kl) \qquad 1)$$

where *i*, *j*, *k*, and *l* are functions in the AO basis and *t* and *u* are from the fitting basis.

The integral formation is parallelized over the fitting basis component of the 3-index quantity. For the RI-SCF algorithm the integrals are generated in row order (AO basis) but used in column order (fitting basis set). As the integrals are computed, the fitting basis index is transformed by using the 2-index 2-center electron repulsion integrals to form the quantities that are used to form the Fock matrix [17]:

$$\langle ij|kl\rangle = \sum_{\tilde{t}\,\tilde{u}} \langle ij|\tilde{t}\rangle V_{tu}^{-1}(\tilde{u},kl) \qquad 2)$$

Consequently, the number of integrals used is reduced from $O(N^4)$ to $O(N^3)$. The Fock matrix construction is done in two components forming the traditional Coulomb and the exchange contributions [16]. The Coulomb contribution requires two straightforward distributed matrix-matrix multiplications with $O(N^3)$ computational work (contraction of the integrals with the appropriate density matrix elements) with $O(N^3)$ integrals read from disk. The exchange contribution does not factorize in the new basis set representation and thus does not have a simple reduction in computational effort [17]. The exchange contribution has the same I/O input requirements but $O(N^4)$ computational effort. The I/O library should allow asynchronous random access to the required integrals at each phase of the computation.

For the RI-MP2 algorithm the AO basis indices (*i*, *j*, *k*, and *l* above) are also transformed to the molecular orbital basis [18]. The integral I/O is $O(N^3)$, and the computational effort is $O(N^4)$ because of the transformation.

Both the RI-SCF and the RI-MP2 algorithms are expressed in terms of array operations. Their disk-based implementations could perform I/O in either independent or collective mode. A collective, array-oriented parallel I/O API is preferred because it allows out-of core algorithms to use high-level array abstractions that correspond to the original mathematical model. These abstractions shield the chemist from having to deal with the intricacies of the underlying hardware and having to use the mundane low-level I/O operations on individual files.

2.4 Checkpoint and Restart

A checkpoint/restart capability is an important feature for many long-running chemistry calculations. In order to stop and restart such calculations, the full state of the program has to be saved and retrieved upon restart from permanent storage. Few scalable parallel computers provide operating system support for this capability. However, parallel I/O libraries can be used to support user-managed explicit checkpoint/restart.

For example, nuclear gradient or Hessian computations of large chemical systems require integral derivatives that number, respectively, 9 or 45 times the number of integrals used for the energy calculation [19].

(This assumes the use of translational invariance for the computation of the first and second order integral derivatives.) These computations are typically done in a direct or semi-direct fashion even on conventional computers, since at the simplest level only a single pass over the integral derivatives is required. The intermediate contributions to the nuclear gradient and Hessian as well as the associated state (i.e., the values of the loops generating the pass over the integral derivatives) could be stored on the disk. This would allow a restart of these expensive computations based on the stored results. Care must be taken to optimize the ratio of the costs of storing the restart information with the overall cost of the computation.

Several implementations of checkpoint/restart are possible in chemistry calculations. In the rather naive approach, following the array distribution in memory, each processor could store a local part of the array in a private/local file. However, this approach has only limited practical utility because of the need to manage hundreds or even thousands of individual files, and the difficulties that arise when restarting on a different number of nodes. Hence, it is preferable to transfer data collectively from in-memory distributed arrays to comparable objects ("disk resident arrays") located on disk. This would allow checkpoint/restart with an arbitrary number of nodes in the restart phase.

3 ChemIO Requirements and Approach

The computational chemistry applications that we surveyed helped us identify three distinct abstractions that generalize the I/O operations and structures used in existing chemistry applications or correspond to specific requests made by developers of new codes. The abstractions are as follows:

1. *Array structures implemented on disk*: Collective I/O operations that transfer data between sections of array data structures distributed over computer memories and equivalent data structures located on disk. These operations may be used in the RI-SCF and RI-MP2 algorithms, for example (Section 2.3), and for checkpoint/restart (Section 2.4).

2. *Private files*: Independent I/O operations in which each processor can be thought of as operating on a distinct file. In this situation, processors never share data via the I/O system, and I/O patterns are too irregular to permit effective application-level collective I/O. The I/O operations are typically used to write "scratch" data, which may be read subsequently during the same run by the generating processor, but does not persist beyond program termination. Private files may be used to store integrals in the traditional HF algorithm, for example (Section 2.1).

3. *Shared files*: Independent I/O operations in which all processors can be thought of as operating on the same file. In this situation, processors can share data via the I/O system, but I/O patterns are too irregular to permit effective application-level collective I/O. Shared file operations are typically used to write "scratch" data, which may be read subsequently during the same run, but does not persist beyond program termination. Shared files may be used to store shared vectors in the MRCI algorithm, for example (Section 2.3).

For each of these abstractions, the ChemIO project has defined three distinct libraries: Disk Resident Arrays, Exclusive Access Files, and Shared Files. These libraries share a common low-level infrastructure but are otherwise independent.

In addition to the specific I/O abstractions, the ChemIO project has addressed issues that involve functionality, capacity, or interface restrictions present in Fortran or I/O systems:

- *Asynchronous I/O.* Asynchronous I/O operations (lacking in Fortran 77 and 90) are supported to allow overlapping of time-consuming I/O with computations. All algorithms described in Section 2 can employ this latency-hiding technique in some phases of the calculations.

- *Large (>2 GB) files.* In order to provide a portable Fortran interface to files larger than 2 GB, the SF and EAF libraries represent file sizes and offset arguments as double-precision size in the Fortran API. Potential users of the library considered this approach preferable to some other alternatives, for example, the Intel Paragon's

extended file interface, which provides a set of library calls to operate on nonstandard (in Fortran) opaque datatypes required to store large values. The availability of a portable Fortran interface to files larger than 2 GB has proved useful in a number of applications, including semi-direct MP-2.

- *Byte-addressable files.* The Fortran direct file access model imposes on applications the requirement that I/O be performed on record units. Records must be of a fixed size that is specified when a file is created or opened. Applications with varying I/O request sizes may address this restriction by transforming data structures to round up request sizes; however, this approach can cause more data than is actually needed to be read or written to disk. Alternatively, the record size may be reduced to the smallest unit (for example, byte/integer/floating-point number) into which requests may be decomposed. While convenient, however, this latter approach tends to result in many small read/write operations and hence low performance. This shortcoming of Fortran direct access files is avoided in ChemIO. The ChemIO files in EAF and SF can be seen as byte-addressable memory, which can be accessed at any location with arbitrary granularity and request size variation.

- *Performance hints.* Important to the usability and perceived performance of the application-oriented I/O libraries is the ease with which library performance can be tuned to support a particular algorithm. In order to optimize performance without sacrificing ease of use, ChemIO supports a system of hints that allow the user to describe important parameters of the I/O algorithm rather than control the low-level parameters (such as data layout on disk, striping, size of internal buffers) of the actual implementation. These hints describe typical request size and shape (in the case of 2-D operations), data type, and approximate size of disk-resident data structures.

4 The ChemIO System

The ChemIO system is designed and implemented in a modular fashion. The three user-level libraries—Disk Resident Arrays, Exclusive Access Files, and Shared Files—are layered on a device library called ELIO (ELementary I/O) that provides a portable interface to different file systems (Figure 2). The DRA, EAF, and SF modules are fully independent: each can be modified or even removed without affecting the others. ELIO itself is not exposed to applications. This approach provides flexibility in selecting optimal implementations, without compromising portability. The design goal was to be able to leverage other state-of-the-art I/O libraries and systems such as MPI-IO [20] when robust and high-performance implementations become available, while preserving the user-level API of ChemIO. The ChemIO libraries have been implemented on distributed-memory computers (Intel Paragon, IBM SP- x, Cray T3D/E, Fujitsu VX/VPP), shared-memory computers (KSR-2, SGI PowerChallenge and Origin-2000), and networks of workstations.

4.1 ELIO Device Library

The ELIO library implements a set of elementary I/O primitives including blocking and nonblocking versions of read and write operations; wait and probe operations to control status of nonblocking read/writes; and file operations including open, close, delete, truncate, end-of-file detection, and an inquiry function for the file/filesystem that returns the amount of available space and filesystem type. Most of these operations are commonly seen in various flavors of the Unix filesystem; ELIO provides an abstract, portable interface to such functionality.

The encapsulation of low-level I/O operations in a separate device library module has several advantages. Reduced maintenance and porting efforts is one benefit. Missing functionality on different filesystems can be implemented without affecting the code in the user-level modules. In particular, nonblocking I/O can be implemented in ELIO on platforms that still do not support it, for example, by using a thread package such as Pthreads or Nexus [21]. ELIO nonblocking I/O is currently implemented on top of Posix AIO and vendor-specific APIs (for example, on Intel PFS). Another benefit of the device library approach is that it allows us to create a set of abstractions better suited than conventional Unix I/O for implementing parallel I/O libraries. For example, in multithreaded programs, the well-known programming difficulties caused by the separation of seek from read and

write operations are alleviated: instead of supporting a separate seek function, ELIO extends read and write operations with an offset argument. Another extension to the otherwise familiar Unix-like read/write interface is the file descriptor that ELIO uses to cache filesystem-specific information. This extension is helpful in programs that use ChemIO under multiple filesystems. For example, a program that runs on an IBM SP can use the ChemIO API to create scratch disk resident arrays on a processor-private filesystem (JFS), permanent disk resident arrays under parallel shared filesystem (PIOFS), and EAF files on the local filesystem.

4.2 Exclusive Access Files

The EAF module supports a particularly simple I/O abstraction in which each processor in a program is able to create files to which it alone has access. The EAF interface is similar to the standard C Unix I/O interface and is implemented as a thin wrapper on the ELIO module. It provides Fortran (and C) applications with capabilities that include

- `eaf_write` and `eaf_read`: blocking write and read operations,
- `eaf_awrite` and `eaf_aread`: nonblocking (asynchronous) write and read operations,
- `eaf_wait` and `eaf_probe`: operations that can be used to control/determine completion status of outstanding nonblocking I/O requests,
- `eaf_stats`: operation that takes a full path to a file or directory and returns the amount of disk space available and filesystem type (e.g., PFS, PIOFS, standard Unix),
- `eaf_length` and `eaf_truncate`: operations that allow us to determine length and truncate file at specified length, respectively,
- `eaf_eof`: operation that determines whether the end of file has been reached, and
- `eaf_open`, `eaf_close`, `eaf_delete`: interface to Unix open, close, and unlink operations.

EAF allows the programmer to think of secondary storage as a byte-addressable single-dimensional array. The traditional seek operation is not supported. Instead, the application specifies file locations via an additional argument (offset) to read and write operations.

The EAF module offers three potential advantages over conventional Unix I/O. First, it provides to Fortran programs (most scientific applications including chemistry have been and continue to be developed in Fortran) additional functionality such as a nonblocking I/O interface (the Fortran 77 and 90 standards both lack this important capability) and access to large files (>2 GB). Second, it enhances portability. Third, the integration of EAF with other ChemIO modules can, in principle, allow common optimizations and the enforcement of operational consistency between I/O operations in different ChemIO library modules.

EAF offers a more general I/O model than that of Fortran 77. There are no fixed record size restrictions or separation of sequential from direct access modes. The byte-addressable disk memory in EAF is more convenient to use and, for variable-sized requests, is potentially faster than direct access mode in Fortran. Nevertheless, EAF performance is competitive with that of Fortran fixed-record direct access files, as shown in Figures 3 and 4. These figures compare the performance of EAF and Fortran direct access files on the IBM SP, Intel Paragon, Cray T3D, and Cray T3E. The IBM SP performance was measured on a system located at Pacific Northwest National Laboratory (PNNL) that contains 512 Power-2 Super nodes and runs AIX 4.2. We used the local disks available on each node rather than PIOFS, since it provides the most scalable environment for EAF. Intel Paragon performance was measured on the 128-node system located at Oak Ridge National Laboratory, which runs OSF 1.4.2 and supports Intel PFS. Cray performance was measured for high-performance filesystems on the 256-node Cray T3D and T3E located at National Energy Research Supercomputing Center (NERSC), running UNICOS

MAX 1.3.0.4 and UNICOS/mk 1.4.2, respectively. We report only relative performance of EAF as compared with direct-access mode in Fortran 77, rather than absolute bandwidth numbers which depend on the particular hardware configurations. The figures compare bandwidth for read and write operations that transferred 80 MB of data in 1.6 kB, 16 kB, 160 kB and 1.6 MB chunks (which in the figures correspond to four bars for each system) with random access to the files. With the exception of the 1.6 kB case on the Cray T3D, EAF outperforms the Fortran model on all the systems. The rate of improvement varies depending on the system and chunk size with the best value exceeding a factor of four on the Intel Paragon.

4.3 Disk Resident Arrays

Many computational chemistry parallel algorithms have been implemented in terms of a shared-memory programming model called Global Arrays (GA). The GA library [5,6] implements a shared-memory programming model in which data locality is managed explicitly by the programmer. This management is achieved by explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. The GA library allows each process in a MIMD parallel program to access, asynchronously, logical blocks of physically distributed matrices, without the need for explicit cooperation from other processes. This functionality has proved useful in computer graphics (parallel rendering), in financial calculations (security value forecasting), and especially in numerous computational chemistry applications. Today, many chemistry programs, totaling over 1.5 million lines of code, use GA (R. Harrison, personal communication), with NWChem [11] alone exceeding 400,000 lines.

The GA model is useful because it exposes to the programmer the non-uniform memory access (NUMA) characteristics of modern high-performance computer systems. Moreover, by recognizing the communication overhead for remote data transfer, the GA model promotes data reuse and locality of reference. The disk resident arrays (DRA) model extends the GA model to another level in the storage hierarchy, namely, secondary storage. It introduces the concept of a disk resident array—a disk-based representation of an array–and provides functions for transferring blocks of data between global arrays and disk arrays. Hence, it allows programmers to access data located on disk via a simple interface expressed in terms of arrays rather than files. The benefits of global arrays (in particular, the absence of complex index calculations and the use of optimized array communication) can be extended to programs that operate on arrays that are too large to fit into memory.

By providing distinct interfaces for accessing objects located in main memory (local and remote) and on the disk, GA and DRA render visible the different levels of the memory hierarchy in which objects are stored. Hence, programs can take advantage of the performance characteristics associated with access to these levels. Recall that in modern computers, memory hierarchies consist of multiple levels, but are managed between two adjacent levels at a time. For example, a page fault causes the transfer of a data block (page) to main memory while a cache miss transfers a cache line. Similarly, GA and DRA allow data transfer only between adjacent levels of memory. In particular, data transfer between disk resident arrays and local memory is not supported. Since we would be jumping between nonadjacent levels in the NUMA hierarchy, performance is expected to be disappointing, and a portable implementation problematic.

Disk resident arrays have a number of uses. They can be used to checkpoint global arrays. Implementations of out-of-core computations can use disk arrays to implement user-controlled virtual memory, locating arrays that are too big to fit in aggregate main memory in disk arrays, and then transferring sections of these disk arrays into main memory for use in the computation. DRA functions are used to stage the disk array into a global array; individual processors then use GA functions to transfer global array components into local storage for computation. If the global array is updated, a DRA write operation may be used to write the global array back to the appropriate component of the disk array. DRA has been designed to support collective transfers of large data blocks. No attempts are made to optimize performance for small (<0.5MB) requests.

We use a simple example to illustrate the use of the DRA library. The following code fragment creates a disk array and then writes a section of a previously created global array (`g_a`) to the larger disk array (see Figure 5). The `dra_init` function initializes the DRA library and allows the programmer to pass information about

how the library may be used and the system on which the program is executing. This information may be used to optimize performance.

```
#include 'dra.fh'
#include 'global.fh'
rc = dra_init(max_arrays, max_array_size, total_disk_space, max_memory)
rc = dra_create(MT_DBL, 2000, 4000,'Big Array','bigfile', DRA_W, -1, -1, d_a)
rc = dra_write_section(.false., g_a,600,1600,300,2300, d_a,750,1750,500,2500,
request)
rc = dra_wait(request)
rc = dra_close(d_a)
rc = dra_terminate()
```

The `dra_create` function creates a two-dimensional disk resident array. It takes as arguments its type (integer or double precision); its size, expressed in rows and columns (2000× 4000); its name; the name of the "meta- file" used to represent it in the file system; mode information (`DRA_W` indicating that the new disk array is to be opened for writing); hints indicating the dimensions for a "typical" write request ("-1" indicates unspecified); and finally the DRA handle (`d_a`). The function, like all DRA functions, returns a status value indicating whether the call succeeded.

The `dra_write_section` function writes a section of a two-dimensional global array to a section of a two- dimensional disk array. In brief, it performs the assignment

```
d_a[750:1750, 500:2500] = g_a[600: 1600, 300: 2300]
```

where `d_a` and `g_a` are a disk array and global array handle, respectively. The first argument is a transpose flag, indicating whether the global array section should be transposed before writing (here, `.false.`). The operation is asynchronous, returning a request handle (`request`); the subsequent `dra_wait` call blocks until termination.

The GA and DRA libraries have been implemented on distributed-memory computers (Intel Paragon, IBM SP- x, Cray T3D/E), shared-memory computers (KSR-2, SGI Power Challenge), and networks of workstations. Disk Resident Arrays are implemented in different ways on different systems. In some cases, a single disk array is stored as a single file in an explicitly parallel I/O system (PIOFS on the IBM SP-x, PFS on the Intel Paragon) or in a sequential filesystem striped implicitly across multiple disks (for example, Cray T3D, SGI); in other cases, a single disk array is striped across multiple disk files in a fashion dependent on the hint information provided by the applications.

DRA has similarities in its design to libraries such as Vesta [22], Panda [23], PASSION [24], and MPI-IO [20], which also support collective I/O operations on arrays. DRA differs from these libraries in its simple array-oriented asynchronous API and its integration with the Global Arrays shared-memory model [25]. To allow more efficient implementation on wide variety of filesystems and disk configuration, and minimize application code complexity, DRA uses hints to determine layout of data on the disk. Other systems such as Panda and MPI-IO expect from the user to specify data layout on the disks. The DONIO [26] library caches a copy of a file in memory distributed across available processors and allows access to the cached file in the noncollective fashion. In this respect it combines some aspects of GA and DRA but provides less control of data locality and movement between different levels in storage hierarchy.

We present microbenchmark results to demonstrate the performance characteristics of the DRA library. These benchmarks use a simple program that performs reads and write operations involving a 5000× 5000 double-precision global array and a 10000× 10000 disk array. Transfers to both aligned and nonaligned sections of the disk array are tested. The disk array is closed and opened between consecutive DRA write and read operations to flush system buffers. We timed a series of calls to obtain average times.

Figure 6 shows measured transfer rates on the 512-node Intel Paragon located at Caltech. The PFS filesystem is implemented on top of 12 RAID disks. The DRA library stores the disk array data in one PFS file opened in `M_ASYNC` mode. We used fixed 2 MB internal DRA buffer and varied the number of processes that perform I/O. We present performance results for operations that reference array sections aligned or nonaligned

with the two- dimensional layout of the data on disk. With nonaligned operations, DRA internally transfers larger blocks of data than those that correspond to the specified arrays sections and therefore, in general, are less efficient than the aligned operations. Aligned write operations on Paragon appear to execute faster than the corresponding read operations when more than one I/O processor is used. In the nonaligned cases, the disk array section involved in the DRA I/O operation was decomposed into aligned and nonaligned subsections. The difference in DRA I/O performance for both aligned and nonaligned I/O transfers is consistent with the two-dimensional data layout on the disk used in the DRA implementation on Paragon and is described in [27]. In general, increasing the size of the DRA internal buffer results in larger PFS I/O requests and in higher achieved bandwidth.

The DRA library achieves 42.7 MB/s for an aligned write from 32 processors with 2 MB internal buffer. A nonaligned write achieves 36.9 MB/s, and aligned and nonaligned reads achieve 35.5 and 34.2 MB/s, respectively. Our results are consistent with I/O rates reported by other researchers using tuned handcoded applications on the same computer. For example, Miller et al. report peak read rates of 35.2 MB/s for a 94 MB transfers and 37.8 MB/s for a 377 MB read transfers [28]; we get 35.5 MB/s for a 200 MB transfer. Thakur et al. [29] report a write bandwidth of 10.85 MB/s from 32 Paragon processors when using the Chameleon I/O library in an astrophysics application. Chameleon uses a fixed internal buffer size of 1 MB.

The SP-1.5 system at Argonne uses the TB-2 switch and Thin 67-MHz Power1 processors running AIX 3.2.5. We obtained peak performance results of 33.4 MB/s for aligned writes, 31.4 MB/sec for nonaligned writes, 44.0 MB/s for aligned reads, and 42.1 MB/s for nonaligned reads from 16 processors with DRA implemented on top of PIOFS. These also appear to be competitive with handcoded I/O programs. Performance results for the implementation based on a collection of local disks (such disks are available on every node of the IBM SP-x), are depicted in Figure 7, and demonstrate almost linear scaling with the number of processors/disks used. The bandwidth in AIX read/write operations to the local disk on this particular system is less than 2 MB/s.

From these results we conclude that the DRA library is able to achieve high I/O rates on parallel filesystems (such as Intel PFS or IBM PIOFS) and on collections of independent disks or local filesystems (such as the JFS on the IBM SP-x).

4.4 Shared Files

The Shared File module is the most recent addition to the ChemIO system. It supports the abstraction of a single contiguous secondary storage address space (“file”) to which every process has access. Processes create and destroy SF objects in a collective fashion; other I/O operations are noncollective. A shared file can be thought of as a one- dimensional array of bytes located in shared memory, except that the library interface is required to access the data.

The following hints can be provided by the user when a shared file is created:

1. Hard limit (not to be exceeded) for file size.

2. Soft limit that corresponds to the best approximation of the shared file size; however, it might be exceeded at run time.

3. Size of a “typical” request.

The hint arguments are optional, meaning that the user can always provide “-1” value which corresponds to the “don’t know” selection. The hints are used by the library to determine the striping factor and other internal optimizations.

Noncollective I/O operations in SF include read, write, and wait operations. Read and write operations transfer the specified number of bytes between local memory and disk at a specified offset. (SF, like EAF, does

not support a separate seek operation.) The library does not perform any explicit control of consistency in concurrent accesses to overlapping sections of the shared files. For example, SF semantics allows a write operation to return before the data transfer is complete, which requires special care in programs that perform write operations in critical sections, since unlocking access to a critical section before write completes is unsafe. To allow mutual exclusion control in access to shared files, an `sf_wait` function is provided that can be used to enforce completion of the data transfer so that the data can be safely accessed by another process after access to the critical section is released by the writing process. An `sf_waitall` function allows the program to wait for completion of multiple SF operations specified through an argument array of request identifiers.

The actual size of a shared file might grow as processes perform write operations beyond the current end-of-file boundary. Data in shared files are implicitly initialized to zero, meaning that read operations at locations that have not been written return zero values. However, reading beyond the current end-of-file boundary is erroneous.

The following code fragment illustrates the use of the SF model. Processes call `sf_create` to create a shared file, specifying a location path and meta-file name, an upper bound for the file size, an estimate for the actual file size (a hint), and a size of a typical request (another hint). A handle to the shared file is returned. Processes then write the data stored in the double precision array `contribution` to a predefined location in the file. After completion of the asynchronous write operation, process 0 reads the content of the entire file into the array `unsorted`. Finally, processes delete the shared file; synchronization is embedded in `sf_create` and `sf_terminate`.

```
double precision contribution(1000), unsorted(10000)
double precision offset, bytes
integer handle, id, rc, myproc, numproc
rc = sf_create('/pfs/shared', 1d6, 1000000d0, 8000d0, handle)
offset = myproc*1000*8d0
bytes  = 1000*8d0
rc = sf_write(handle, offset, contribution, id)
rc = sf_wait(id)
call barrier()
if(myproc.eq.0) then
rc = sf_read(handle, 0d0, dble(numproc*1000*8), unsorted, id)
rc = sf_wait(id)
endif
call sf_destroy(handle)
```

Shared files can be used to build other I/O abstractions. In many cases, this process requires adding an additional consistency control layer. A single file pointer view, for example, can be implemented by adding an atomically modifiable pointer variable located in shared memory, by using the GA toolkit (atomic `read_and_increment` operation) or some other means.

We present a performance study of Shared Files on the 256-node IBM SP-2 at Lawrence Livermore National Laboratory. The SP-2 system is equipped with the TB-3 switch and compute nodes are Thin-2 66-MHz Power2 processors running AIX 4.1.4. The PIOFS parallel filesystem uses eight server nodes, each rated at 8MB/s. In the benchmark, processors read and write independently at different nonoverlapping locations in a shared file stored under the IBM PIOFS or the IBM PFS. The size of request is constant in the given run, and so is the file size (approximately 1.3 GB), regardless of the number of processors used. Processes write data into shared file starting from the file beginning in round-robin fashion. Immediately after global synchronization (the library does not support close or flush operation), processes read the data from the shared file. In order to avoid reading data cached in local memory by the OS, any individual processor reads different portions of shared file from those it wrote. Because of the lack of close or flush operations in the SF semantics and the nature of this benchmark, we expected that both in our benchmark and in the actual applications write operations would perform somewhat better than reads. The benchmark measures aggregate bandwidth achieved as a function of the number of processors performing simultaneous I/O operations to the same file.

We present performance results for 32 kB and 64 kB request size in Figure 8. They represent the average performance on the non-dedicated system with no other applications using PIOFS. The SF is able to exploit full

potential of the PIOFS with 16 processors (twice the number of PIOFS servers) and 64 kB request size reading and writing to the shared file. With the smaller request size, the performance is significantly lower and we see larger difference for read and write performance than for the 64 kB case. Additional experiments with 128 kB and 256 kB request size indicated that the maximum aggregate bandwidth could not be improved over the performance level achieved with 64 kB request size. The overall performance of Shared Files is very good (64 kB case), and with a larger number of processors (representing high-level contention in access to shared file) there is no significant performance degradation. Note that our benchmark represents a worst-case scenario: I/O operations in actual applications would be interleaved with computations, and therefore only a fraction of processors would access shared file at the same time.

5 Applications

The ChemIO project has provided three distinct and complementary I/O models that address the needs of computational chemistry algorithms. These models form a foundation for developing I/O algorithms in chemistry. We expect that different ChemIO abstractions might be more or less effective on different platforms depending on the I/O hardware and system configuration. For example, the traditional or semi-direct Hartree-Fock algorithms implemented on top of EAF might be more scalable than algorithms implemented on top of Shared Files on systems that provide local disk for every node in a parallel computer, for example IBM SP. On other systems, such as the Intel Paragon, with high-performance filesystems optimized for large, collective I/O operations rather than independent I/O performed to hundreds or thousands of individual files, algorithms based on the DRA model are expected to be the most competitive. The alternative I/O models in ChemIO support the development of "I/O poly-algorithms" in which the most efficient algorithm and suitable I/O model can be chosen to match the system capabilities.

ChemIO components have already been used to develop several large computational chemistry applications, and several other codes are under development. We describe experiences with four large chemistry applications.

5.1 Experience with Hartree-Fock Method

We describe experience with the Hartree-Fock method discussed in Section 2.1. In particular, we concentrate on effectiveness and tradeoffs in choosing disk-based approaches with respect to the system I/O architecture. As an example, we ran three versions of the Hartree-Fock method to solve a fluorinated biphenyl system $C_{14}H_8F_6$ on the IBM SP at Pacific Northwest National Laboratory and the Cray T3E-600 at NERSC. The IBM SP at PNNL contains local disks on every node and has the PIOFS parallel filesystem connected to every node. However, on the SP we used local disk rather than PIOFS because it provides the most scalable platform for the EAF model. On the Cray T3E, which does not provide local disk on every node, we used a filesystem (shared by all nodes) which contains 52 SCSI disks.

The three versions of HF algorithm we tested are

1. traditional method that stores all the integrals on disk,

2. semi-direct (hybrid) method that stores only some integrals (more expensive) on disk and recomputes the others, and

3. direct method, which does not use disk at all and recomputes all integrals as needed.

It is straightforward to compare the traditional algorithm with the direct algorithm. The former is purely a function of the aggregate disk bandwidth to the secondary storage device, and the latter is a function of the computational power of each node in the massively parallel processing (MPP) system. For a given MPP supercomputer, if the aggregate bandwidth does not scale with the number of processes, the direct algorithm clearly will be appropriate for large processor counts, since the computational power does scale. On an IBM SP, where the EAF I/O model is scalable because of the local disk on each processor, the direct:traditional ratio is

1.5:1.0 on 64 nodes for a 324 basis function case in which $5x10^8$ integrals are calculated. This test case on the IBM SP system (64 nodes) produced a measured bandwidth for read operations on a per node basis of 8.0 MB/s. For a 32-node calculation, the measured per node bandwidth was 8.1 MB/s. These values demonstrate the obvious scalability of the local disk configuration on the IBM SP systems.

By comparison, a CRAY T3E system at NERSC for the same test case yielded a direct:traditional ratio of 3.9:1.0 on 64 nodes. The measured bandwidth for read operations on a per node basis for 64 nodes was 0.58 MB/s. For 32 nodes the measured T3E bandwidth was 0.94 MB/s. These values lead to the conclusion that the EAF model is less scalable on the T3E.

A semi-direct algorithm can be tuned to the strengths of an MPP system. For example, a system that has scalable I/O and memory (e.g., the IBM SP) can use all available I/O and memory resources to cache integrals and can recompute only those that do not fit into the combined disk space and aggregate memory. This algorithm on the same 324 basis function test case had a ratio of direct:semi-direct of 2.9:1 on 64 nodes of an IBM SP (2.2:1 on the T3E system). In fact, on large numbers of processors, the semi-direct algorithm can produce superlinear speedup (R. Harrison, personal communication). Tuning the semi-direct algorithm to the strengths of both the IBM SP and the Cray T3E yielded results that are less than 15 percent different in the time to solution of the SCF algorithm. This fact shows the strength of the adaptability of the algorithm for MPP supercomputer systems.

5.2 Experience with RI-MP2 and RI-SCF

The approximate "resolution to the identity" second-order many-body perturbation theory method (RI-MP2) uses a combination of two- and three-center integrals to approximate the four-center two-electron integrals. Although the overall computational cost of RI-MP2 is $O(N^5)$, the algorithm is well suited for massively parallel implementation [18]. Bernholdt and Harrison implemented RI-MP2 using the GA toolkit and local/private files to store transformed integrals. Their algorithm uses $O(N^2)$ main memory and $O(N^3)$ disk storage—a significant improvement over the requirements of exact MP2. The integrals are stored in main memory (for smaller problems) or written to disk with each process writing a subset of integrals to its own file. An important computational step in this method is a matrix multiplication that uses the integral data and is done in chunks sized to fit in main memory. When the RI-MP2 implementation was developed, the ChemIO tools were not available; instead, simple Fortran reads and writes were used for each process. This API required integral data to be read by the same processors that wrote them, in turn contributing to the complexity of the load-balancing algorithm. The access mechanisms available in the DRA allow a more straightforward load-balancing algorithm in the RI-MP2. Recently, this algorithm has been converted to use the DRA library to utilize its higher I/O efficiency and asynchronous operation in order to overlap in-core matrix multiplication with I/O, thus in some sense implementing an out-of-core matrix multiplication.

The DRA and GA libraries were used to develop a parallel implementation of the RI- SCF method [17]. The work of Frühtl et al. [17] and Bernholdt and Harrison [18] has shown the utility of the DRA model. The ability to collectively read logical subblocks of data into globally distributed arrays abstracted at an application programmer interface in DRA has made the implementation of RI-SCF and RI-MP2 chemistry applications more straightforward. The asynchronous operations in ChemIO also were found to be useful. Experience with DRA shows that, in both RI-based applications, when asynchronous access is available, the disk I/O becomes less of a bottleneck. Currently, the bulk of the program time is localized in the distributed matrix multiply. In other words, the computational work is still larger than required disk I/O, at least for the large-scale RI-based algorithms.

5.3 Experience with MRCI

In order to alleviate the physical memory limitations in the Multi-Reference Configuration Interaction calculation, ChemIO's shared file mechanism recently was adopted [30] in the COLUMBUS program [14]. Files are accessed in a noncollective fashion, with individual processors reading and writing records containing compressed data. Data is read from disk, uncompressed, updated, compressed, and written back to the disk. Since the update can

affect the compression rate, the size of the data written to the disk might be different from that of the original data. The program stores expansion and product vectors in a compressed form in an one-dimensional global array. At the beginning of a calculation, the amount of memory needed to store all these vectors in a compressed form is unknown. The program allocates all possible distributed core memory for one-dimensional global array. During the execution, if global array is not large enough, the program uses ChemIO shared files to continue the calculation.

From the application perspective, the performance of the disk- based approach was found to be very good on the IBM SP-2 with Shared Files implemented on top of PIOFS. As expected, the SF version was slower than the memory-based implementation, but its scaling properties appeared to be very good. An entire accumulation process—reading compressed data, decompression, accumulation (atomic daxpy operation), compression, and finally writing the compressed date—took about 3 MB/s per processor when the data was stored in the distributed arrays in main memory (global arrays). With the shared files, the rate was about 2 MB/s per processor and almost independent of the number of processors (which was in the 16 to 96 range). With this level of performance, the CI calculations based on the SF model become feasible for very large problems, those where the compressed storage of the CI Hamiltonian and trial vector exceed the aggregate memory of the system, see Section 2.2.

6 Conclusions

The ChemIO project was formulated with the two related goals of facilitating the use of high-performance parallel I/O in computational chemistry codes and advancing understanding of the interfaces and techniques required to support parallel I/O. We believe that the project has been successful in both respects. An analysis of the I/O requirements of a range of computational chemistry codes led us to define a broad set of I/O interfaces supporting exclusive access files, disk resident arrays, and shared files. Implementations of these interfaces were developed for scalable parallel computers, and shown to be efficient in microbenchmarks. Finally, the practical utility of these interfaces and implementations was demonstrated by their successful adoption in a range of computational chemistry applications. The Disk Resident Arrays interface, in particular, appears to represent a significant contribution to parallel I/O technology, providing a clean integration of a distributed shared-memory programming model with parallel I/O. Our future research plans are twofold. First, we will explore the value of the I/O interfaces in additional applications. Second, we will address implementation issues that arise on next-generation scalable parallel computers and I/O systems. Possible extensions that we are considering include: 1) support for higher dimensional (>2) disk resident arrays when they become available in the GA library, 2) adding read-ahead and write-behind capabilities to EAF and SF, and 3) providing additional hints in the API to exploit the application characteristics.

Acknowledgments

The EAF component of ChemIO was originally implemented by Brian Toonen. Jace Mogill (ANL) and Robert Harrison (PNNL) contributed to the more recent implementations of ELIO and EAF. Work at ANL was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the Scalable I/O Initiative, a multi-agency project funded by DARPA, DOE, NASA, and NSF. Work at PNNL was performed under the auspices of the High Performance Computing and Communications Program of the Office of Computational and Technology Research, U.S. Department of Energy under contract DE-AC066-76RLO1830 with Battelle Memorial Institute which operates the Pacific Northwest National Laboratory. Access to the CRAY T3E was provided by the Massively Parallel Processing Access Program of the National Energy Research Scientific Computing Facility from time granted by the Division of Mathematical, Information, and Computational Sciences, Office of Computational and Technology Research, U.S. Department of Energy, and the Division of Chemical Sciences, Office of Basic Energy Sciences, Office of Energy Research, U.S. Department of Energy. Access to the Intel Paragon at Oak Ridge National laboratory was provided by the High Performance Computing and Communication Initiative Grand Challenge Program for computational chemistry funded by MICS/OTR/DoE. Access to the Intel Paragon operated by Caltech Institute of Technology on behalf of the Concurrent Supercomputing Consortium was provided by Pacific Northwest National Laboratory.

References

[1] A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry*. McGraw-Hill Publishing Company, New York, NY, 1989.

[2] M. F. Guest, R. A. Kendall, J. A. Nichols, J. T. H. Dunning, and E. M. S. Gordon. Challenges of high performance computing in chemistry. Technical Report PNL-10202, Pacific Northwest National Laboratory, March 1995. Available from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

[3] R. J. Harrison and R. L. Shepard. Ab initio molecular electronic structure on parallel computers. *Ann. Rev. Phys. Chem.*, 45:623–658, 1994.

[4] R. A. Kendall and M. F. Guest. Input and output in chemistry applications. In A. M. Tentner, editor, *Grand Challenges in Computer Simulation, High Performance Computing 1995, Proceedings of the 1995 Simulation Multiconference*, pages 522– 526. Society for Computer Simulation, 1995.

[5] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A portable 'shared-memory' programming model for distributed memory computers. In *Proceedings of Supercomputing 1994*, pages 340–349, 1994.

[6] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high- performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[7] A. T. Wong and R. J. Harrison. Approaches to large-scale parallel self-consistent field calculations. *Journal of Computational Chemistry*, 16:1291, 1995.

[8] R. Shepard. Elimination of the diagonalization bottleneck in parallel direct-SCF methods. *Theoretica Chimica Acta*, 84:343, 1993.

[9] J. Almlöf, J. K. Faegri, and K. Korsell. Principles for a direct SCF approach to LCAO-MO ab initio calculations. *Journal of Computational Chemistry*, 3:385–399, 1982.

[10] J. Almlöf and P. R. Taylor. Computational aspects of direct SCF and MCSCF methods. In C. E. Dykstra, editor, *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules*, pages 107–125. E., Reidel, Dordrecht, Germany, 1984.

[11] D. E. Bernholdt, E. Apra, H. A. Fruchtl, M. F. Guest, R. J. Harrison, R. A. Kendall, R. A. Kutteh, X. Long, J. B. Nicholas, J. A. Nichols, H. L. Taylor, G. I. Fann, R. J. Littlefield, and J. Nieplocha. Parallel computational chemistry made easier: The development of NWChem. *International Journal of Quantum Chemistry Symp.*, 29:475, 1995.

[12] R. A. Kendall and R. J. Harrison. A parallel version of ARGOS: A distributed memory model for shared memory unix computers. *Theoretica Chimica Acta*, 79:337–347, 1991.

[13] M. Häser and R. Ahlrichs. Improvements of the direct SCF method. *Journal of Computational Chemistry*, 10:104–111, 1989.

[14] H. Dachsel, H. Lischka, R. Shepard, J. Nieplocha, and R. J. Harrison. A massively parallel multireference configuration interaction program - the parallel COLUMBUS program. *Journal of Chemical Physics*, 18:430, 1997.

[15] P. R. Taylor. Integral processing in beyond-Hartree-Fock calculations. *International Journal of Quantum Chemistry*, 31:521– 534, 1987.

[16] O. Vahtras, J. Almlöf, and M. W. Feyereisen. Integral approximations for LCAO-SCF calculations. *Chemical Physics Letters*, 213(5, 6):514, 1993.

[17] H. A. Früchtl, R. A. Kendall, R. J. Harrison and K.G Dyall. A parallel implementation of RI-SCF on parallel computers. *International Journal of Quantum Chemistry*, 64, 63, 1997.

[18] D. E. Bernholdt and R. J. Harrison. Large-scale correlated electronic structure calculations: The RI-MP2 method on parallel computers. *Chemical Physics Letters*, 250:477, 1996.

[19] M. Dupuis and H. F. King. Molecular symmetry. gradient of electronic energy with respect to nuclear coordinates. *Journal of Chemical Physics*, 69(9):3998–4004, 1978.

[20] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NAS, NASA Ames Research Center, Moffett Field, CA, January 1995. Version 0.3.

[21] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[22] P. Corbett and D. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[23] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. Supercomputing '95*, December 1995.

[24] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[25] Y. Chen, I. Foster, J. Nieplocha, M.Winslett, Optimizing Collective I/O Performance on Parallel Computers: A Multisystem Study, *Proc. 11th ACM Intl. Conf. on Supercomputing*, ACM Press, 1997.

[26] E. F. D'Azevedo and C. Romine. DONIO: Distributed object network I/O library. Report ORNL/TM-12743, Oak Ridge National Lab., 1994.

[27] J. Nieplocha and I. Foster. Disk Resident Arrays: An array-oriented library for out-of-core computations. In *Proc. Frontiers of Massively Parallel Computation*, pages 196–204, 1996.

[28] C. Miller, D. Payne, T. Phung, H. Siegel, and R. Williams. Parallel processing of spaceborne imaging radar data. In *Proceedings of Supercomputing '95*. ACM Press, 1995.

[29] R. Thakur, W. Gropp, and E. Lusk. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with special emphasis on Parallel Databases and Parallel I/O*, pages 24–35. Lecture Notes in Computer Science 1127. Springer-Verlag., September 1996.

[30] H. Dachsel. personal communication. PNNL, 1997.

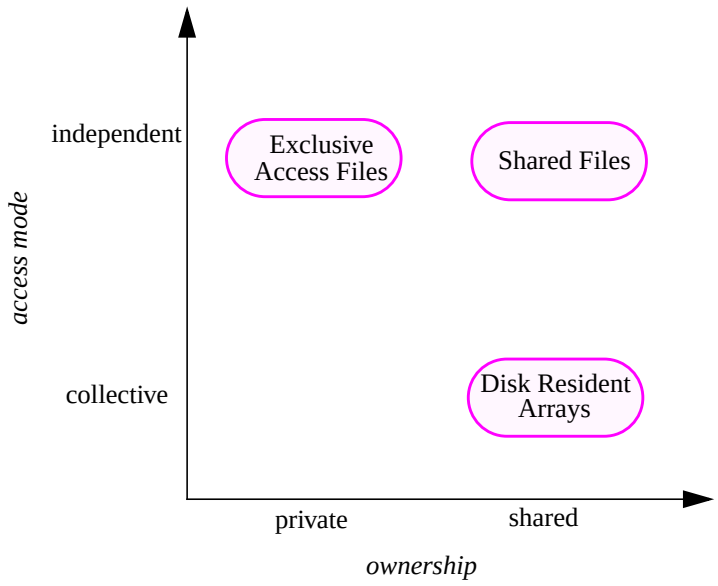

Figure 1: Comparison of access modes and file ownership in ChemIO modules

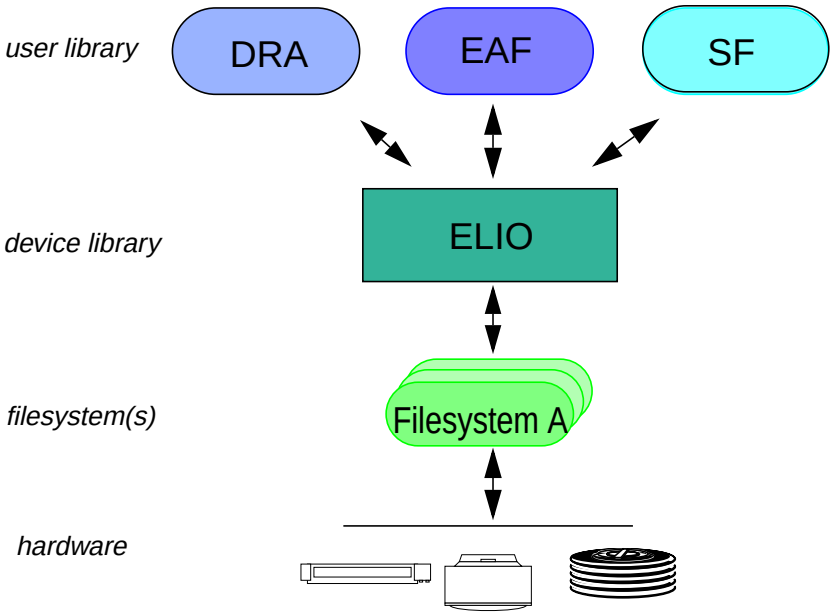

Figure 2: Structure of ChemIO system

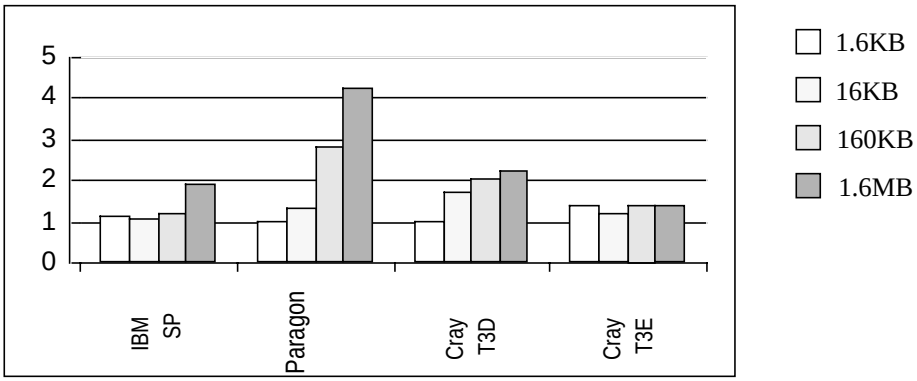

Figure 3: Relative performance of EAF and Fortran write operations as function of the record size

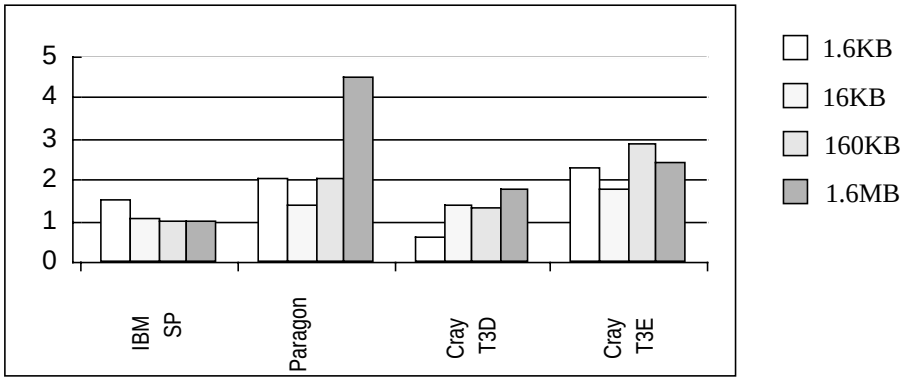

Figure 4: Relative performance of EAF and Fortran read operations as function of the record

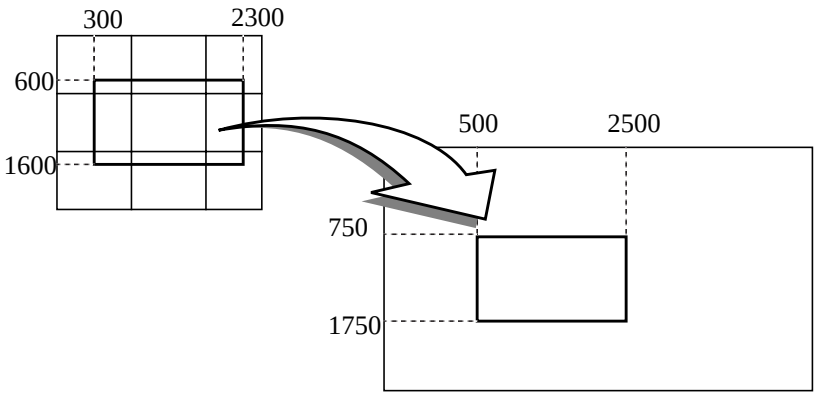

Figure 5: Writing a section of a distributed array to a disk resident array

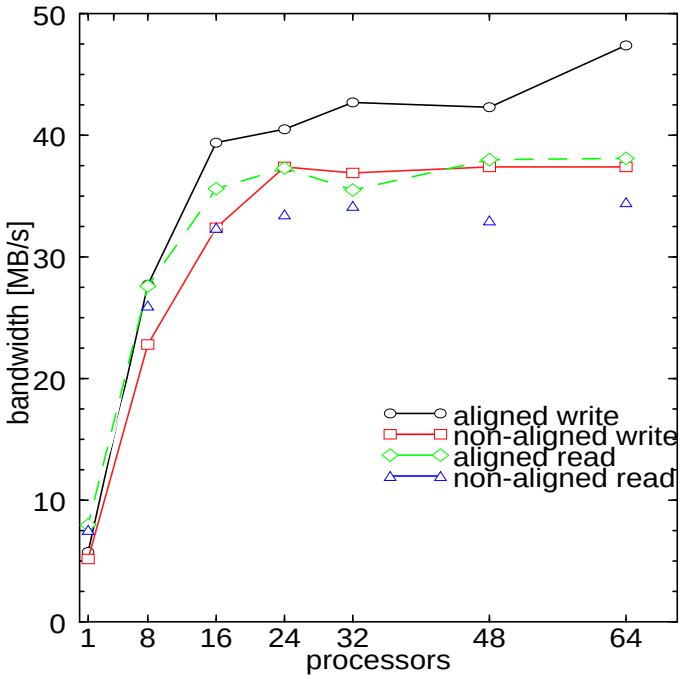

Figure 6: Performance of DRA on Intel Paragon

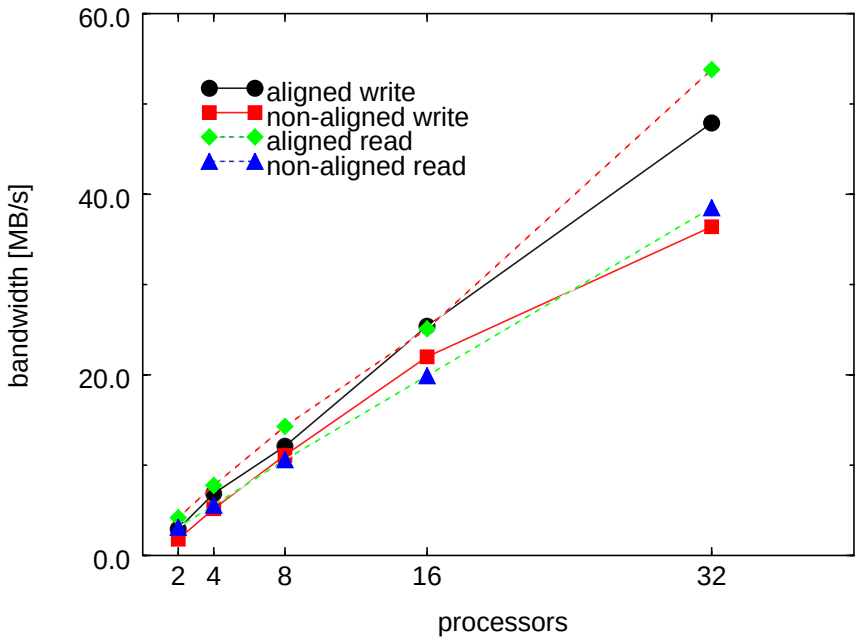

Figure 7: Performance of DRA on local disks of the IBM SP-1.5

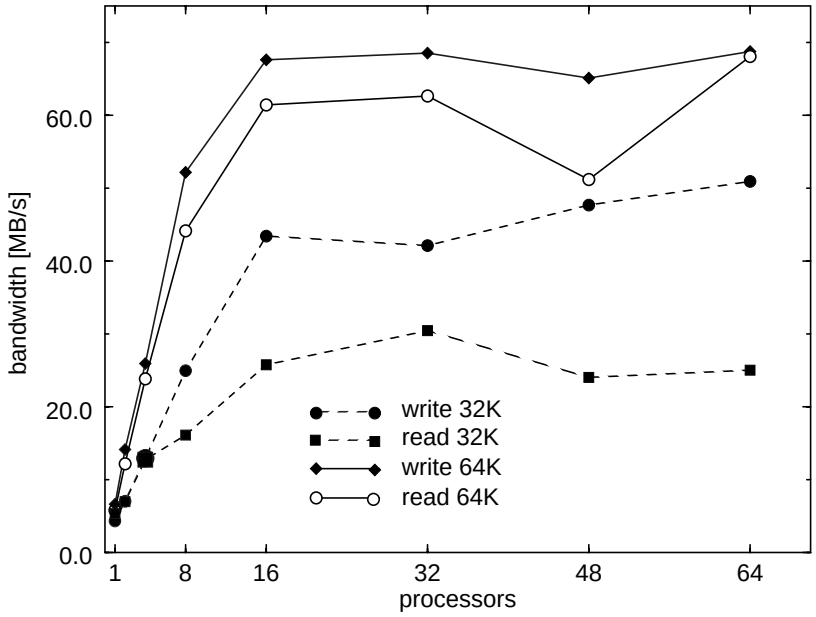

Figure 8: Performance Shared File operations as function of request size and number of processorson on the SP-2