An out-of-core implementation of the COLUMBUS massively-parallel multireference configuration interaction program

Holger Dachsel Jarek Nieplocha Robert Harrison

Pacific Northwest National Laboratory

Abstract

In this paper, we describe a novel parallelization approach we developed to solve the largest multireference configuration interaction (MRCI) problem ever attempted. From the mathematical perspective, the program solves the eigenvalue problem for a very large, sparse, symmetric Hamilton matrix. Using an out-of-core approach, shared memory programming model, improved data compression algorithms, and dynamic load balancing we were able to solve a problem six times larger than previously reported. The potential curve for the chromium dimer was calculated with a Hamilton matrix of dimension 1.3 billion (1,295,937,374). This task involved moving 1.5 terabytes of data between main memory and secondary storage per MRCI iteration. Furthermore, by employing Active Messages and user-level striping to combine multiple files on local disks on the IBM SP into a single logically-shared file, the execution time of the program was reduced by a factor of three, as compared to our initial implementation on top of the IBM PIOFS parallel filesystem.

Introduction

The multireference configuration interaction (MRCI) method [1] is widely used in computational chemistry to obtain accurate predictions of properties of chemical systems. The MRCI method is arguably the most generally-applicable, high-accuracy method for determination of the electronic structure of small molecules. It can be used to predict the energetics and other properties of molecular systems from which may be derived chemical information including thermochemistry, spectroscopy and detailed models of chemical reactions. Since the computational expense scales as greater than the sixth power of the molecular system size, it is only applicable to small molecules. The general applicability of MRCI results in the program being significantly more complex than other computational chemistry programs. However, an efficient, massively-parallel implementation of the MRCI method is highly desirable to enable greater accuracy and also application to molecular larger systems. Within the COLUMBUS program package [1-3] only the MRCI program has been parallelized, but for high-accuracy calculations this is by far the most expensive step. The parallelization of this program [4] has been conducted by a collaboration involving researchers at Pacific Northwest National

Laboratory, Argonne National Laboratory, and the University of Vienna. Over a period of about seven years, the parallel efficiency of the program has increased from a paltry 50% on about 8 nodes of an Intel distributed-memory computer, to over 94% on 512 nodes of a CRAY-T3E. Most of this improvement in scalability results from extensive algorithmic modifications to expose greater parallelism, eliminate overhead from the parallel algorithm compared to the sequential one, eliminate disk I/O, and reduce communication and memory usage in part by using data compression. Until now the problem size that can be treated with this code [4] was, however, limited by the aggregate available memory to problems not much larger than have been done for some time using out-of-core techniques on vector supercomputers. Described in this paper is our implementation in the COLUMBUS MRCI program of an out-of-core approach that performs well on the IBM SP massively parallel computer. Since the program is implemented on top of portable programming tools and libraries it can be used on other platforms. We demonstrate the capability of this program by application to a problem six times larger than any previously reported.

The MRCI method approximates the electronic wavefunction (at a given nuclear geometry) with a linear expansion in a basis of n-electron functions or configurations. The energy is then minimized by variation of the coefficients, or, equivalently, by the iterative extraction of the lowest few eigen-pairs of the Hamilton matrix. The dimension of the Hamilton matrix (number of configurations) can become very large and even though the matrix is very sparse, it cannot be stored explicitly. Instead, matrix-vector products are computed from integrals (*(k|l)* and *(kl|mn)* below) and coupling coefficients (*A* and *B* below) by using the Graphical Unitary Group Approach (GUGA) [6].

$$H_{ij} = \sum_{k}^{N}\sum_{l}^{N} A_{ij}^{kl}(k|l) + \sum_{k}^{N}\sum_{l}^{N}\sum_{m}^{N}\sum_{n}^{N} B_{ij}^{klmn}(kl|mn)$$

The mostly dense integrals must be stored and the sparse coupling coefficients recomputed as required. The sparse matrix-vector product is reformulated as many, small (O(100)), dense matrix-matrix operations which execute with high single-processor efficiency. Most of the complexity in both the sequential and parallel algorithm arises from avoiding unnecessary movement of data (which include the integrals, input vector and output product vector), or redundant computation of coupling coefficients, while maintaining a completely general algorithm that is applicable to a wide class of chemical problems. Problems specific to the parallel algorithm are discussed below. The number of integrals is proportional to the fourth power of the one-electron basis set used in the calculation, typically a few hundred, so there may be several GBytes of these.

The iterative Davidson diagonalization method [5] is used to solve the eigenproblem. In this scheme, the most time consuming step is the computation of the sparse matrix-vector product of the Hamilton matrix and the expansion vector. A set of expansion vectors is used to project the original orthogonal eigenvalue problem to a subspace in which a small non-orthogonal eigenvalue problem is solved.

Current state of the art MRCI calculations are in the range between 100 and 200 million configuration state functions [7,8]. Our goal was to develop a program which allows calculations of more than one billion configurations.

Parallel Algorithm

The most computationally demanding section of the MRCI algorithm is the diagonalization of the Hamilton matrix. Therefore, we concentrated on this step in our parallelization work. In an early version of the code, we used message passing to parallelize the computation of the matrix-vector product. Using the replicated data model each processor had to store a local copy of the expansion and corresponding product vector. This model required distribution of the expansion vector at the beginning and a global sum at the end. The computations of the subspace Hamilton matrix and the overlap matrix were not carried out in parallel. Due to the replicated data model the calculation size was limited by the memory of just a single processor. Furthermore, the standard message-passing model did not provide sufficient functionality to distribute fully data structures. The calculations would be performed in a MIMD task-parallel fashion driven by data-dependent dynamic load balancing. A major improvement could only be achieved if asynchronous access to data distributed over the core memory of all processors were possible in the sense of a "virtual shared memory". In this way two major bottlenecks could be removed at once:

- broadcasting of all data at the beginning of each Davidson iteration and collecting the results via a global sum could be avoided because all necessary data were available just in time and
- memory inefficiencies of the replicated data approach would be avoided. The problem would be limited now by the aggregate memory of the entire parallel computer.

We use for this purpose the Global Array (GA) toolkit [9] that supports one-sided communication capabilities in context of distributed arrays. To address needs for a scalable locking mechanism to protect access to the distributed shared data structures in MRCI, we added to the GA toolkit distributed lock operations. In addition to global arrays we still have the option to store multiple copies of certain data files (like the two-external integrals) locally in core because these files are relatively small but frequently accessed. We have developed a special data management scheme called virtual disk to store compressed and uncompressed data locally. The key advance described in this paper is to allow the largest shared data structures (the virtual disks) to spill over onto physical disk storage, while still maintaining the one-sided data access model, efficient scalable parallel execution and the ability to cache data in physical memory.

Virtual disk

A virtual disk is used to store variable length records of different logical files in an arbitrary sequence in private/ shared disk/memory. Files that are too large to fit into memory may spill over onto disk. Each variable length record in a file is divided into a linked list of blocks. In the absence of compression, these subblocks are of constant length, but when compression is enabled the dynamic updating of the data requires that each block is fragmented and also managed as a linked list. To avoid frequent access to physical disk, the record headers are cached in memory. However, if even these are too large, all record information for a logical file can be written to the virtual disk except for the address of the first block of each record which may be straightforwardly computed. If compression is enabled, the data structure of the virtual disk provides all the functionality necessary to store the data compactly. The aim is to have a memory saving almost equal to the saving coming from compression. As soon as a data segment is not fully filled the space is freed. Runtime statistics have shown that up to 99% efficiency is obtained in practice. Another goal was that the virtual disk should have no more than 10% space overhead required by the information to maintain the nested linked lists. Virtual disks are implemented on top of any or all local files or memory (for private disks), or global arrays or shared files (for shared virtual disks). The Shared File library is described below.

Mutual exclusion is necessary to maintain the virtual-disk data structures in shared files. Atomic read-and-increment operations suffice to manage the list of free blocks used to grow/shrink the file. However, if the fragmentation of compressed data changes the mutual exclusion becomes necessary around the updating of the linked list information of that record. The operations used are described in more detail below.

Data management

The major bottleneck of the CI calculation is the large amount of data to be transferred. Due to memory limitations the expansion and product vectors are segmented. Four segments, two expansion vector segments and two product vector segments have to be kept in core to compute the matrix vector product. The expansion vector has to be read about the number of segments times, so there is a trade-off between increasing the available parallelism and increasing the total volume of data movement. Updating the product vectors requires double the amount of data to be transferred. Additionally, data compression has been implemented in connection with the implementation of a shared file (described in the following section). Major improvements have been made to increase the efficiency of the earlier compression algorithm [4]. The compression scheme for the expansion vectors as well as the compression scheme for the product vectors are now adjusted dynamically. Information about the structure of the eigenvectors is used to eliminate the redundant data completely. For better scalability, a time sorted task list has been implemented to improve the dynamic load balancing.

The diagonalization task consists of two subsections, the subspace manipulation and the matrix-vector product. Due to memory limitations all expansion and product vectors are segmented equally. The algorithm of the matrix vector multiplication requires four segments to be stored in core. Each segment is divided into eight records. The vectors are stored in record units.

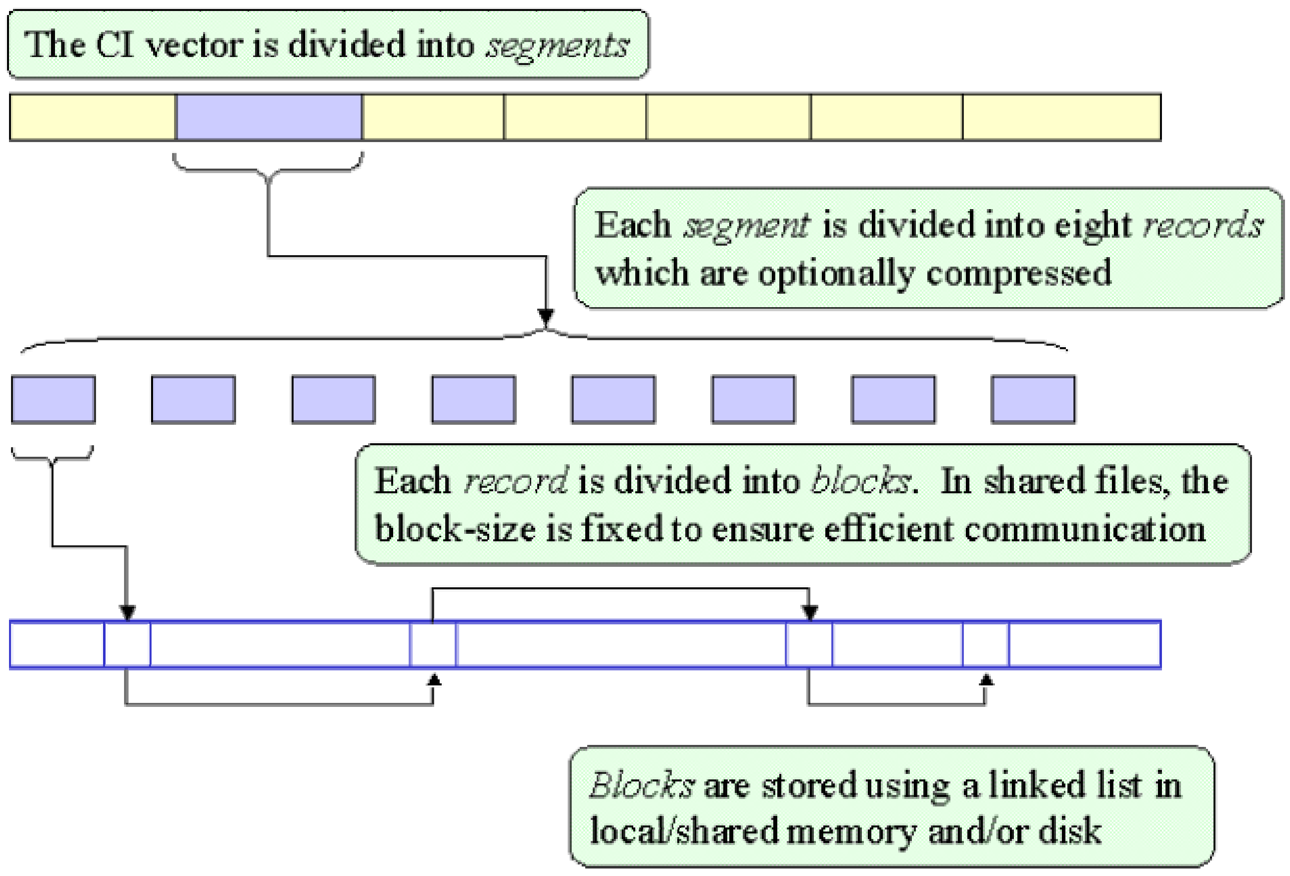
The CI vector is divided into *segments*
Each *segment* is divided into eight *records* which are optionally compressed
Each *record* is divided into *blocks*. In shared files, the block-size is fixed to ensure efficient communication
Blocks are stored using a linked list in local/shared memory and/or disk

Figure 1: CI vector structure

The program allocates all possible distributed main memory for a one-dimensional global array and then creates a shared file to store data that does not fit in the distributed main memory, see Figure 1. Secondary storage is 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 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 average I/O request size in this program is approximately 30 KB. Due to the vector segmentation, the amount of available in-core memory, and the high efficiency of the compression, the request size could not be easily increased.

Additional, mutual exclusion is necessary when accumulating into a virtual shared disk. For scalability, there is a separate lock for each record.

Data Compression

A data compression method [4] has been developed to significantly reduce the amount of data by eliminating data that will not contribute to a given matrix-vector product for the requested convergence threshold. The compression is based upon an error analysis for the expansion and product vector elements which includes the required solution accuracy and current convergence status. The requirement for the elements of the expansion and product vectors not to exceed given constant overall errors means that the number of significant digits in these elements varies and this number will increase or decrease with variation in the absolute values of the vector components. To make use of this fact, we devised a floating-point representation with a variable length mantissa, and discarded the insignificant digits. Major improvements have been made with regard to the previous algorithm [4]. Information about the structure of the eigensystem is now used to adjust the compression scheme dynamically, also taking into account the possibility of root crossings.

The efficiency of the compression depends upon the sparsity of the Hamilton matrix and the requested convergence threshold. It is important to note that the number of iterations is not increased by the compression scheme when compared with an equivalent calculation with the same convergence threshold, but without compression. For an eigenvalue threshold of 10^{-6}, saving factors of between 5 and 30 have been seen for various MRCI calculations.

Interprocessor Communication

Motivated by the irregularity of communication patterns resulting from the dynamic load-balancing, the shared-memory programming rather than the message-passing model was selected for this application. The portable Global Array (GA) toolkit [9] supports an object-based shared-memory programming paradigm that matches well the communication requirements of this application. By eliminating the need to coordinate the sender and receiver of the data the programming effort was greatly reduced.

The GA has been traditionally supported on the IBM SP through the interrupt-receive functionality (*rcvncall*) available in the IBM Message Passing Library (MPL). During the course of the work on parallelization of the MRCI code, a new implementation of GA has been developed based on the new low-level communication library called LAPI. The development of LAPI and the optimized port of GA to the IBM SP was a subject of a collaborative project between IBM and PNNL that started in March of 1996 [12]. At present time, both the PowerPC SMP and RS/6000 uniprocessor models of the SP support LAPI [12], a commercial implementation of Active Messages (AM) [13]. In particular, LAPI provides:

- *active message* operations,
- *put*, and *get* remote memory copy operations, and
- additional synchronization and ordering operations useful in the shared memory style of programming.

The LAPI implementation of GA has improved latency and bandwidth of this system, especially for codes like COLUMBUS that use 1-dimensional rather than multi-dimensional distributed arrays. The primary GA communication operations used in the MRCI code are *get*, *put*, and *accumulate*. The latency and bandwidth (measured for 2MB requests) performance differences between MPL and LAPI implementations of these operations on system equipped with 120MHz P2SC uniprocessor nodes and the TB-3 network adapter in a synthetic microbenchmark are shown in Figure 2.

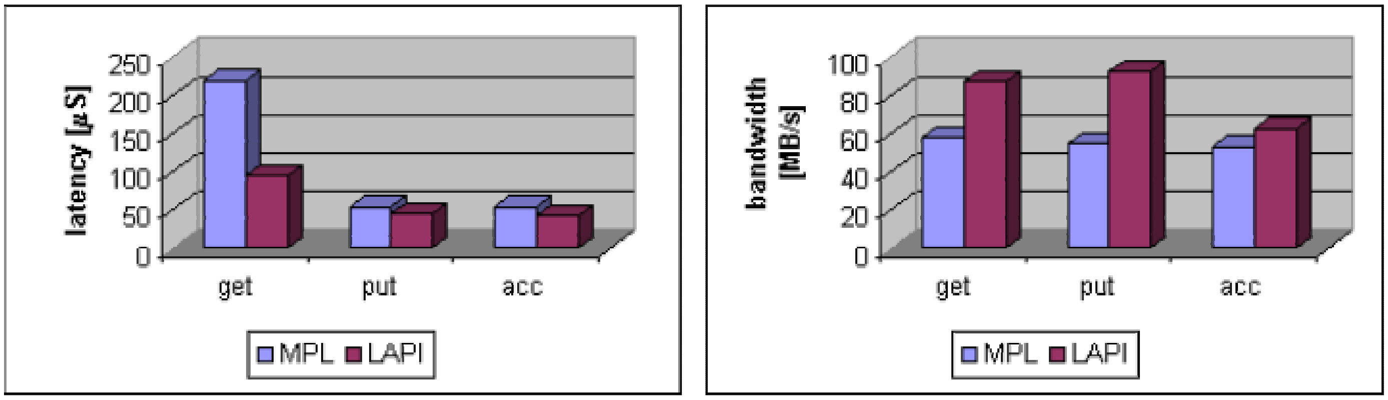

Figure 2: Performance of GA get, put, and accumulate under LAPI and MPL

To support the application needs for mutual exclusion when updating data structures (e.g., *product vector*) on disk or in memory atomically, the lock operations have been added to the GA package. The implementation is platform specific. On systems with shared memory or global address space (Cray T3D/E) the traditional algorithm based on atomic fetch-and-add (already available in GA as *ga_read_inc* operation) and lock queue was used. For distributed memory systems (IBM SP, Intel Paragon), a new algorithm has been implemented that maintains a lock queue on the remote node within the interupt-receive or active message handlers. The cost of acquiring an uncontested lock on the IBM SP is comparable to that of the *get* operation.

Parallel I/O

To address the memory limitations in solution of large problems, the MRCI code adopted a disk-based approach using the Shared Files library [10]. The portable Shared Files (SF) library is one of the three parallel I/O libraries developed by the ChemIO project [10] at Pacific Northwest and Argonne National Laboratories. The ChemIO project has defined I/O interfaces that capture the I/O patterns found in important computational chemistry applications and provided high-performance portable implementations of these interfaces. The Shared Files library supports the concept of a parallel file in which every process in a parallel computation can read and write independently at arbitrary locations. The differences between SF and other systems include the following:

- Shared files are not guaranteed to be persistent. Shared file operations are typically used to write "scratch" data, which may be read subsequently during the same run, but might not persist beyond program termination. Persistency is a property of the filesystem on which SF is created, rather than the model itself.

- Shared Files support files larger than 2 GB even in the Fortran API.
- Shared Files *read* and *write* operations are nonblocking and contain an offset argument rather than a separate seek operation.
- Unlike record based I/O models in Fortran, SF provide byte addressable disk access.

On the IBM SP, the original implementation of SF depended on the IBM parallel filesystem, PIOFS. The initial results of experiments with this application on the LLNL IBM SP-2 with 8 PIOFS servers were very promising, as described in reference [10]. However, the performance and scalability of SF on the next generation of machine (IBM SP) with faster network, processors and 44 PIOFS servers turned out to be rather disappointing. To address the performance and scalability problems as experienced in the SF-PIOFS implementation of Columbus, we decided to investigate other approaches. Each node of the IBM SP contains at least one local disk. The aggregate I/O bandwidth and capacity of the local disks presented an attractive opportunity to move the shared files from PIOFS to local disks by striping a single logically shared file on local disks available on the computing nodes on which the application was running. However, to support shared memory style of computations involving non-collective access to data stored in a single shared file, a new remote disk access technology had to be developed. We decided not to use the existing approaches like IBM Virtual Shared Disk (VSD) since they:

1. require file system reconfiguration which is not practical in a general purpose multi-application system operation mode adopted by the computing centers supporting a scientific/technical user community,
2. use server processes and communication approaches (based on polling) designed for the traditional client-server applications which likely would consume significant memory and CPU resources and have negative impact on the application process sharing the same processor.

What was needed was a dynamic *part-time-server/part-time-client* I/O model that would localize I/O to the set of resources available to the application during its run-time and had minimal impact on the performance of the application thread. To support the non-collective I/O model of Shared Files we developed the *Distant I/O* model [11]. Distant I/O combines one-sided communication with I/O to secondary storage memory at remote processors. Distant I/O has several useful properties, including the following:

- Distributed view of secondary storage: Secondary storage is used as an extension of main memory in distributed memory systems and accessed with the convenient one-sided communication paradigm.
- Flexibility: DIO can be used to implement parallel I/O models and libraries even on systems that lack parallel/shared filesystems.

- Capacity and bandwidth scalability: As the number of application processors with attached disks grows, the aggregate I/O bandwidth and aggregate capacity proportionally increases.

The distant I/O implementation on the IBM SP uses LAPI Active Messages to send specifications of *read/write* operation to remote nodes. Upon arrival an interrupt is generated, the AM completion handler is invoked and executed by a separate thread. Within the AM completion handler code, LAPI remote memory copy and standard C language library I/O are used. For *dio_write*, *LAPI_Get* transfers data to an internal DIO buffer. This step is followed by a blocking write. For *dio_read,* blocking read is followed by a *LAPI_Put,* which transfers data read from the disk to the internal DIO buffer and then to the user buffer at the requesting processor, see Figure 3. In addition, *LAPI_Put* increments the counter variable *cntr* to notify the requesting node that the data arrived into the user buffer.

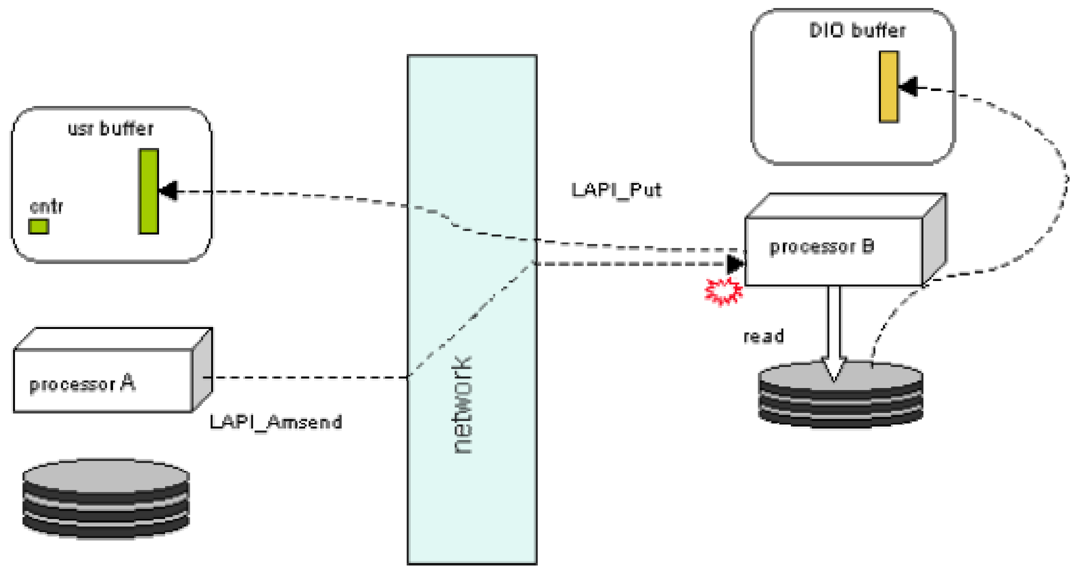

Figure 3: Implementation of remote read operation (*dio_read*) on top of LAPI

Other active-message□style facilities, including the Berkley/Cornell AMs, could be used in a similar fashion to implement DIO on other platforms, including networks of workstations.
The microbenchmark results for distant I/O implemented on local SCSI disks (each rated at approximately 6 MB/s) of the IBM SP at PNNL indicate very good performance of this model, see Figure 4. In this benchmark, we are comparing performance of *read* and *write* operations to local and remote disks with variable request size and total 1 GB of data. The performance differences in access to local and remote files is moderate and decreases with increased request size.

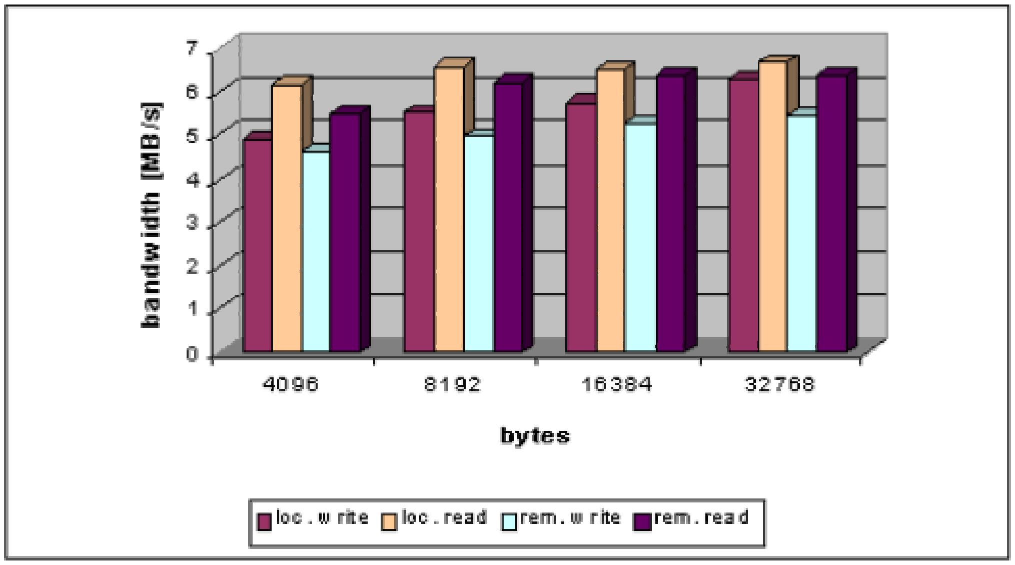

Figure 4: Performance of DIO read and write operations as a function of request size in access to local and remote files

The DIO-based version of the Shared Files library is implemented by striping a parallel logical file across multiple physical files located on all disks available on the computing nodes on which the parallel application is running. It uses Distant I/O to perform remote read/write operations. When a shared file is created, the user can provide several hints that the underlying implementation can exploit to optimize the performance. In particular, the "typical request size" can be used by SF to determine the appropriate value of the striping factor. The striping is performed in a round-robin fashion, see Figure 5.

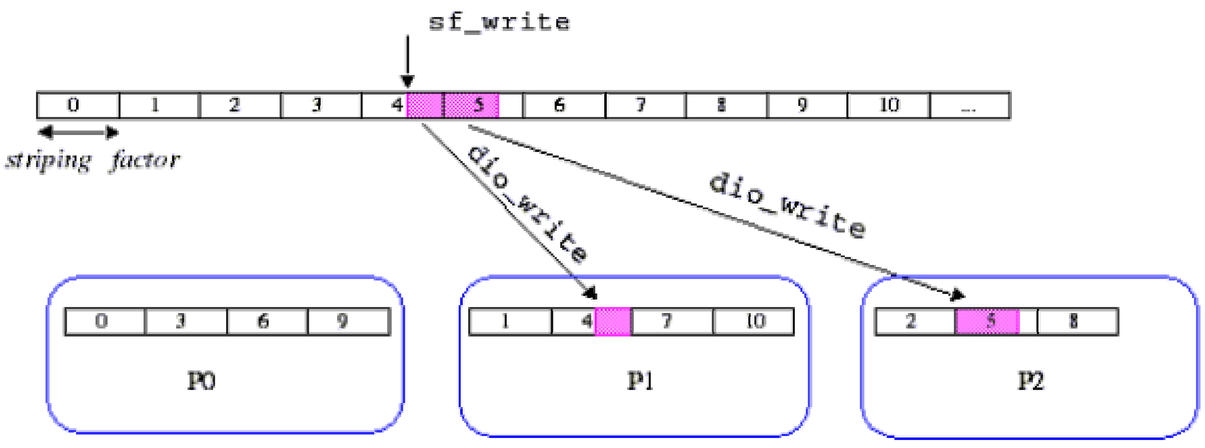

Figure 5: Example sf_write request decomposed into DIO operations on component files for a shared file implemented on three processors with local disks

The SF library is able to support some shared files on PIOFS and others on DIO/local disks in the same program. The particular implementation is chosen dynamically based on the path argument to the `sf_create` operation that specifies the location in a filesystem where a shared file metafile is created. Process 0 creates a file in the specified location and then the other processes attempt to access it. This attempt is successful if the file resides on a shared filesystem, in which case the parallel filesystem implementation is chosen, otherwise the DIO implementation is selected.

Performance Results and Discussion

The calculations we describe in this section were carried out on the PNNL/EMSL IBM SP with 512 Power-2-Super processors most of them equipped with 128 Mbytes of memory. The peak performance on each processor is 480 Mflops. We describe results of the largest MRCI calculation that was ever performed. In addition, we present:

- interprocessor communication performance in context of our application for MPL and LAPI
- scalability study of the application for a smaller problem size that could be performed on 64 processors
- performance of I/O using SF on top of PIOFS and DIO

We have chosen the chromium dimer as one of our benchmark examples because it is an outstanding chemical problem and requires extremely extended CI expansions. Up to this time, a really satisfactory calculation for this system had not been performed. The reason for these large expansion spaces is that the core-valence correlation has to be included in order to give reliable potential energy curves. Moreover, large basis sets (up to *h* functions) are necessary. A series of extended test calculations have been performed so far in order to evaluate the importance of different factors (valence and core-valence electron correlation, basis set effects, computational efficiency, etc.). Both of the calculations used a primitive basis 20s15p10d6f contracted as 4+5s,2+6p,1+6d,1+4f (the notation being core+valence) due to Bauschlicher and Partridge [14], the 3088 CSF reference set of Stoll and Werner [15], and the MR-QAQCC method of Szalay and Bartlett [16]. In the first calculation only the 12 valence electrons are correlated, giving rise to 90,679,216 CSF in D_{2h} symmetry.

In the largest MRCI calculation ever performed, 24 electrons are correlated in an attempt to describe core-valence correlation effects, giving rise to 1,295,937,374 CSF in D_{2h} symmetry. This is a very much harder calculation than a full-CI calculation of similar dimension since the wavefunction is much more complex, and the Hamiltonian is much less sparse.

Interprocessor communication

To better understand performance of MPL and LAPI implementations of GA in the application context rather than a synthetic microbenchmark, we instrumented the application and measured the time required to read the integrals in Columbus. The request size was 262,144 bytes. Figure 6, compares average performance of the MPL and LAPI implementations of GA for this message size. The LAPI version runs three times faster.

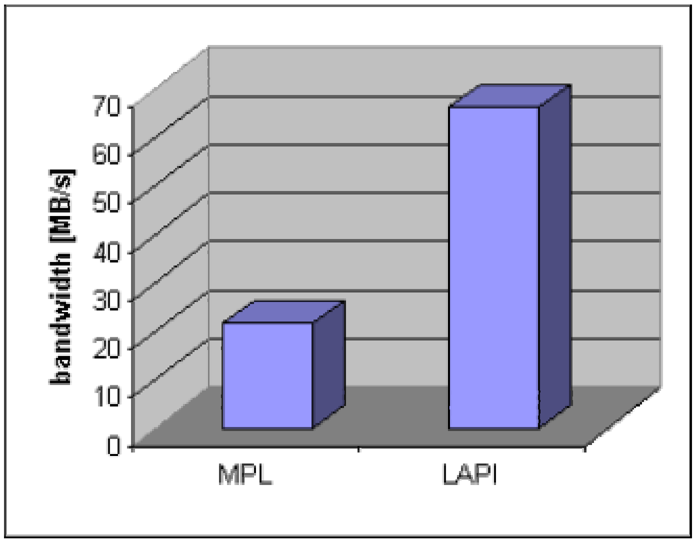

Figure 6: Communication bandwidth in Columbus when reading integrals

Scalability Study

We have chosen a smaller calculation to investigate the scaling of the in-core version of the program. A test case was chosen for which all data could be kept in core starting with 64 processors. The dimension of the Hamilton matrix is 125 million (125,033,696). The relative speedup was calculated with respect to the 64-processor run and is presented in Figure 7. It shows that program scales very well with the number of processors.

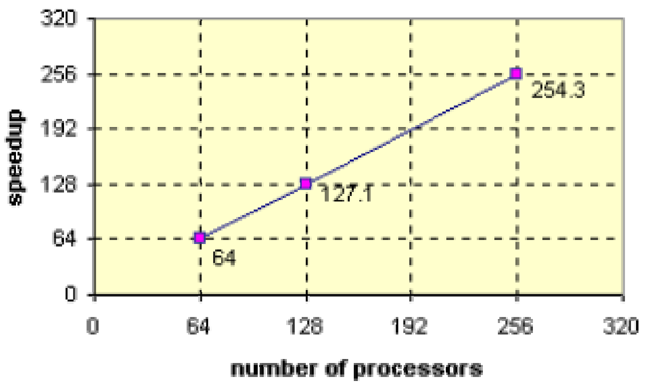

Figure 7: Speedup for a smaller problem

Large calculation

We used both PIOFS and DIO implementations of Shared Files to solve the largest MRCI single and double excitation problem ever attempted, represented by a matrix of dimension 1,295,937,374. The calculations were performed on 128 processors of the 512-node IBM SP with Power-2-Super processors at PNNL. It took 24 iterations to compute one point of the potential curve for chromium dimer. To get the full potential curve, 8 points need to be calculated. Therefore, we ran the program 8 times.

To better illustrate the challenging nature of this calculation, in Figure 8 we show the aggregate volume of data that had to be transferred in a single iteration of the algorithm in the three most data intensive components of the code. Since integrals are frequently used and fit in-core, reading integrals involved the MPL or LAPI communication. The expansion and product vectors are kept in secondary storage. Therefore, the amount of data that had to be moved between main (distributed) memory and secondary storage is approximately 1.5 TB. Due to the dynamic nature of the algorithm (calculations driven by dynamic load balancing) the amounts of data transferred in MPL-PIOFS and LAPI-DIO versions of the code are slightly different.

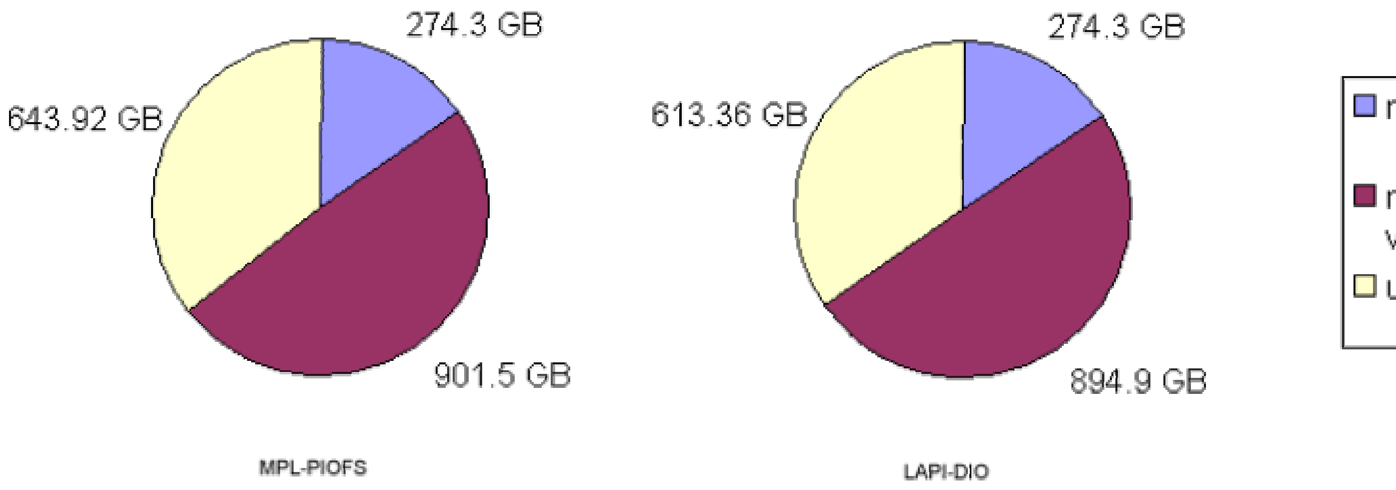

Figure 8: Volume of data (in gigabytes) transferred in different computational phases of the large calculation (per single iteration)

Because of the data volume, despite the significant differences in performance between MPL and LAPI, as shown in Figure 6, the performance of the I/O subsystem has much higher impact on the overall performance of the out-of-core version of the program.

In Table 1, we report timing results for the initial implementation that used SF on top of PIOFS and the newest implementation of SF using DIO on top of local disks. The PIOFS parallel filesystem was configured with 44 servers with 4 SSA disks each (which appears to be the largest PIOFS configuration available at that time anywhere). Each of the compute nodes of the system contains a local SCSI disk rated at 6 MB/s. We report performance of the I/O only for reading expansion vectors (approximately 900 GB/iteration) because the performance was easier to measure and analyze. On the other hand, the updating of the product vector involved: reading, writing, compression and mutual exclusion operations which complicated the performance measurements. For example, the *write* operation due to the buffering and caching effects in AIX could not be timed reliably without impacting the program performance. This was considered unacceptable.

The improvement in the execution time after incorporating the DIO implementation of Shared Files and LAPI implementation of Global Arrays into the program is remarkable. Most of the 70% reduction of the execution time as compared to the initial MPL-PIOFS implementation is contributed to the increase of the Shared File performance. This improved by more than a factor of three by switching to DIO and local disks. With this version, the time spent for I/O has been reduced to about 20% of the program execution time.

Table 1: MRCI performance on top of PIOFS and DIO (local disks) on the IBM SP

	Aggregate I/O time/iteration (reading integrals)	I/O bandwidth/CPU/iteration (reading integrals)	Total execution time (wall clock)
PIOFS	906297.07s	0.994MB/s	305.5 hours
DIO	235708.32s	3.823MB/s	79.6 hours

Our additional scalability study for MRCI indicates that the I/O bandwidth per processor observed in the local disk implementation of MRCI is approximately constant with the number of processors (and disks) used. The observed bandwidth to PIOFS does not scale that well.

Conclusions

Our novel, out-of-core, massively-parallel MRCI algorithm has enabled much larger scale calculations than have been possible before. Calculations with several billion configuration state functions are now quite feasible. The implementation upon the IBM SP using the Shared Files library on top of the Distant I/O model employing the LAPI active message library to support access to physically distributed disks has proven very scalable. In addition, we also obtain close to the hardware limits for I/O bandwidth. Since the program is implemented on top of portable programming tools and libraries such as Global Arrays and Shared Files it can be used on other platforms as well. The substantial progress we achieved has required intimate collaboration over an extended period between computational chemists and computer scientists and led to advancements both in the computer science area (such as the novel Distant I/O model) and chemistry. Future goals include abstracting the essential content of some of the data compression algorithms and distributed data structures, and expressing them as libraries that may be more widely used. Finally, as the gap between the speed of processors and the speed of memory, and especially disk, widens the MRCI algorithm must be adjusted to accommodate the deeper memory hierarchy and proportionately slower disk.

Acknowledgments

This work was supported through the U.S. Department of Energy by the Mathematical, Information, and Computational Science Division phase II grand challenges of the Office of Computational and Technology Research, DOE 2000 ACTS project, and used the Molecular Science Computing Facility in the Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

References

1. H. Lischka, R. Shepard, F. Brown, and I. Shavitt, Int. J. Quantum Chem., vol 15, 91 (1981).
2. R. Ahlrichs, H.-J. Böhm, C. Erhard, P. Scharf, H. Schiffer, H. Lischka, and M. Schindler, J. Comp. Chem., 6, 200 (1985).
3. R. Shepard, I. Shavitt, R. M. Pitzer, D. C. Comeau, M. Pepper, H. Lischka, P. G. Szalay, R. Ahlrichs, F. B. Brown, and J. G. Zhao, Int. J. Quantum Chem., S22, 149 (1988).
4. H. Dachsel, and H. Lischka, *Theor. Chim. Acta*, vol 92, 339 (1995).
5. E. R. Davidson, J. *Comp. Phys.*, vol. 17, 87 (1975).
6. I. Shavitt in The unitary group for the evaluation of electronic energy matrix elements, J. Hinze, Ed., Springer-Verlag, Berlin, , p. 51, 1981.
7. St Evangelisti, G. L. Bendazzoli, R. Ansaloni, and E. Rossi, *Chem. Phys. Lett.*, vol 233, 353 (1995).
8. T. J. Martinez, and E. Carter, Chem. Phys. Lett., vol 241, 490 (1995).
9. J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomputing*, 10:197□220, 1996.
10. J. Nieplocha, I. Foster, R.A. Kendall, ChemIO: High-performance parallel I/O for computational chemistry applications, to appear in *Int. J. Supercomp. Apps. High Perf. Comp.* vol. 12, no. 3, August 1998.
11. J. Nieplocha, I. Foster, H. Dachsel, Distant I/O: One-Sided Access to Secondary Storage on Remote Processors, *Proceedings of the IEEE Symp. High Performance Distributed Computing HPDC-7,* pages 148-154, 1998.
12. G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison , R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, Performance and experience with LAPI □ a new high-performance communication library for the IBM RS/6000 SP. *Proceedings of the International Parallel Processing Symp*osium *IPPS□98*, pages 260-266, 1998.
13. D. Culler, K. Keeton, C. Krumbein, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. Generic active message interface specification. Technical report, University of California at Berkeley, 1994.
14. C.W. Bauschlicher, S. Partridge, *Chem. Phys. Lett.*, vol 231, 277 (1994).
15. H. Stoll and H.J. Werner, *Mol. Phys.*, 88, 793 (1996).
16. P. Szalay and R.J. Bartlett, *J. Chem. Phys.*, 103, 3600 (1995).