Distant I/O:
One-Sided Access to Secondary Storage on Remote Processors

Jarek Nieplocha[†]
<j_nieplocha@pnl.gov>

Ian Foster[‡]
<foster@mcs.anl.gov>

Holger Dachsel[†]
<holger.dachsel@pnl.gov>

†Pacific Northwest National Laboratory
Richland, WA 99352

‡Argonne National Laboratory
Argonne, IL 60439

Abstract

We propose a new parallel, noncollective I/O strategy called Distant I/O that targets clustered computer systems in which disks are attached to compute nodes. Distant I/O allows one-sided access to remote secondary storage without installing server processes or daemons on remote compute nodes. We implemented this model using Active Messages and demonstrated its performance advantages over the PIOFS parallel filesystem for an I/O-intensive parallel application on the IBM SP.

1 Introduction

Recent advances in low-latency, high-speed network technology coupled with inexpensive commodity processors have greatly improved cost-effectiveness of parallel computing. Disk technology also has been advancing rapidly especially with respect to the storage density, capacity and bandwidth. I/O, on the other hand, remains a major bottleneck in many parallel applications, such as climate modeling, computational chemistry, and computational fluid dynamics. We argue that more flexible communication protocols in the style of Active Messages [1] or VIA [2] are needed to exploit inexpensive local storage systems available on single-processor or symmetric multiprocessor distributed and clustered systems. We propose a new parallel I/O model called Distant I/O (DIO) and demonstrate its advantages in a large computational chemistry application.

Parallel I/O systems such as IBM PIOFS, Intel PFS, and PIOUS [3] use a traditional client-server model in which some processors with attached disks act as I/O servers, while other processors are used as compute nodes and I/O clients. This model can work well in dedicated (single-application) environments if the I/O subsystem characteristics match the needs of the particular application. In the multi-application environment, the I/O system is a shared resource which usually impacts the I/O performance delivered the simultaneously running applications. The client-server dichotomy is not required, however, in collective I/O [4]. For example, systems such as Panda [5] and DRA [6] can effectively exploit all local disks on the compute nodes during a collective I/O operation. Nevertheless, although collective I/O is effective for some applications, it is not appropriate for others that require more dynamic and independent (noncollective) access to secondary storage. Certain computational chemistry applications have such access characteristics, for example [7].

We propose a new noncollective I/O strategy called distant I/O (DIO) that targets clustered computer systems in which disks are attached to compute nodes. This hardware configuration is currently supported by networks of workstations (NOWs) and by multicomputers (e.g., the IBM SP) and is likely to remain popular in the future because of the economic and physical attributes of such architectures. DIO supports one-sided access to storage on remote nodes that execute the same user application; hence, all processors can participate in a computation and operate as both I/O clients and (if they have attached disks) servers. A DIO implementation relies for performance on the existence of a single-sided communication protocol able to initiate an I/O operation on a processor that may be performing other computation. Lightweight communication protocols such as Active Messages and Fast Messages [8] can be used for this purpose, or we can use more portable multithreaded messaging systems such as Nexus [9]. Such facilities allow a DIO implementation to avoid the complexities and overheads associated with the server processes and daemons required to support a client-server–based approach.

A primitive Distant I/O mechanism can be used as a building block for higher-level parallel I/O libraries and systems. In this paper, we demonstrate how this can be accomplished by describing a DIO-based implementation of the Shared Files (SF) library that was originally developed

by the ChemIO project [10,7]. The SF library allows a user to define "shared files" that can be accessed and updated independently by any processor in a parallel computation. On the IBM SP, our DIO-based SF implementation outperforms a comparable implementation based on the IBM PIOFS parallel file system by a wide margin, when used by a large computational chemistry application, namely COLUMBUS, a multireference configuration interaction code [11].

The rest of this paper is organized as follows. In Section 2, the DIO model and implementation on the IBM SP are described. In Section 3, a parallel shared file implementation using DIO is presented. Section 4 contains microbenchmark performance results for DIO. Section 5 compares the performance of a large application running on top of DIO with the performance of PIOFS on the IBM SP. Finally, Section 6 presents our conclusions.

2 Distant I/O Model and Architecture

One-sided communication allows a process in a distributed-memory system to access data residing in the address space of another process, without the explicit cooperation of the second process. 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:

- *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. Furthermore, such systems can be customized (for example, by setting striping factors and caching policies) to match the needs of a parallel application, rather than relying on system-wide settings.
- *Capacity and bandwidth scalability*: As the number of application processors with attached disks grows, the aggregate I/O bandwidth and aggregate capacity proportionally increases.

In order to explore the practical utility of Distant I/O, we have defined a DIO application programmer interface (API) and constructed an implementation of this API on the IBM SP. The API is based on the C run-time library, but extends it in four areas:

1. Instead of a separate seek operation, offset is used as an additional argument to read/write operations.
2. The file descriptor is replaced by a handle that references a file on a local or remote processor.
3. A "home" process/processor id is added to identify the location of the file.
4. A request handle is added to support nonblocking versions of calls.

For example, the following are the DIO counterparts to the standard *read* and *write* operations:

```
dio_read(handle, offset, buffer,
    bytes, proc,request)

dio_write(handle, offset, buffer,
    bytes, proc, request)
```

Nonblocking DIO functions are designed to allow overlapping of remote I/O operations with other activities. Such operations are completed by calling the *dio_wait* function, which takes as an argument the request handle returned by the corresponding *dio_write/read* operation. The remote file handle is a local representation of the remote file descriptor. It can be obtained by registering a file for DIO access either through a collective DIO operation similar to the MPI-2 *MPI_Win_create* [11] or through a directory service.

2.1 Distant I/O on the IBM SP

We discuss the architecture of DIO by describing its implementation on the IBM Scalable POWERparallel (SP) system. This massively parallel computer employs as building blocks processor nodes similar to the IBM RS/6000 workstation. Every node contains at least one local disk based on SSA or SCSI technology. The nodes are connected through a high-speed network that supports 38 μsec one-directional latency and 100 MB/s bandwidth in the point-to-point user-space communication [13]. In addition, some larger SP configurations support an optional parallel filesystem, PIOFS. PIOFS is striped on multiple disks connected to dedicated PIOFS server nodes.

The current generation of the IBM SP supports LAPI [13], a commercial implementation of Active Messages [1,14] that provides Active Messages, put, and *get* one-sided remote memory copy operations. LAPI is a multithreaded system compatible with Pthreads, which are supported by the IBM AIX 4.2 operating system on the IBM SP.

Our DIO implementation on the IBM SP uses LAPI Active Messages (AM) to send specifications of read/write operation to remote nodes. Upon arrival, the AM completion handler is invoked and executed by a separate thread. Within the handler code, LAPI remote memory copy and Unix 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 *LAPI_Put*, which transfers data read from the disk to the internal DIO buffer and then to the user buffer at the request-

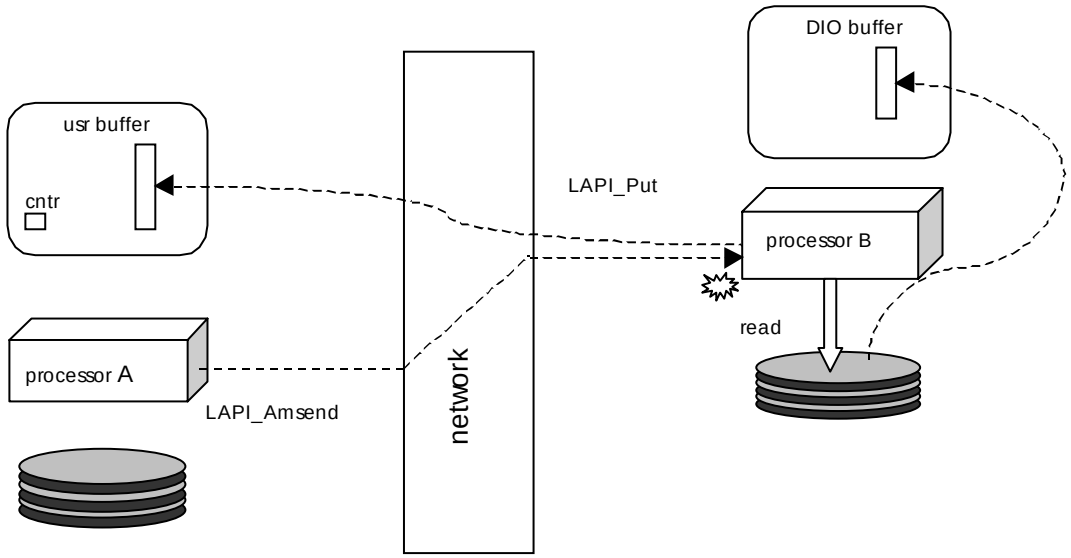

Figure 1: Implementation of dio_read

ing processor. This process is illustrated in Figure 1 for *dio_read.*

The IBM SP implementation of DIO nonblocking operations exploits Posix asynchronous I/O (AIO) when a DIO operation references a file on the local disk. If the data is on a remote disk, the operation returns when an active message is sent to the remote processor. The AM completion handler is executed by a separate thread, which is activated by the LAPI dispatcher when a message arrives [13]. This thread executes concurrently with the main (application) thread and makes the blocking I/O read/write call. When the I/O is completed and the completion handler finishes, LAPI implicitly sends a low-level control message to the requesting processor, which then increments the AM completion counter (*cntr* in Figure 1). This optional feature is enabled when a non-NULL completion counter address is specified in the *LAPI_Amsend* interface. If the data is on a remote file, the *dio_wait* operation waits until this counter is incremented, a process that occurs when the completion handler and I/O operation it executes are completed. If the data is on a local file, the *dio_write* simply calls its Posix AIO counterpart.

The LAPI-based implementation of DIO can be used by MPI programs. However, it cannot be used by applications that rely on MPL, the original IBM proprietary message-passing library which is still available on the IBM SP. The reason for this incompatibility is that the message delivery infrastructure of MPL is based on signals. The most recent implementation of MPI and LAPI both use threads. Unlike MPL, MPI is available in two implementations: signal and thread-based. In our experience, the thread-based implementation of MPI is competitive with the signal-based implementation of that library.

We note that although the counter mechanism in LAPI AM interface is convenient, it is not mandatory to implement DIO. Other active-message–style facilities, including the Berkley/Cornell AMs and Illinois FM, could be used in a similar fashion to implement DIO on other platforms, including networks of workstations. In particular, notification of I/O completion can be accomplished with an explicit message sent back to the requesting node.

3 Parallel Shared Files

We used the Distant I/O model to construct an implementation of Shared Files (SF), a parallel I/O library we had developed before [7]. This library supports the concept of a parallel file with every process in a parallel computation being able to read and write independently at arbitrary locations. This disk access model is similar to the parallel files created in the M_UNIX mode on the Intel PFS and the default parallel file mode supported by the IBM PIOFS. The differences between SF and these other systems include the following:

- Shared files are not guaranteed to be persistent. Persistency is a property of the filesystem on which SF is implemented, rather than the model itself.

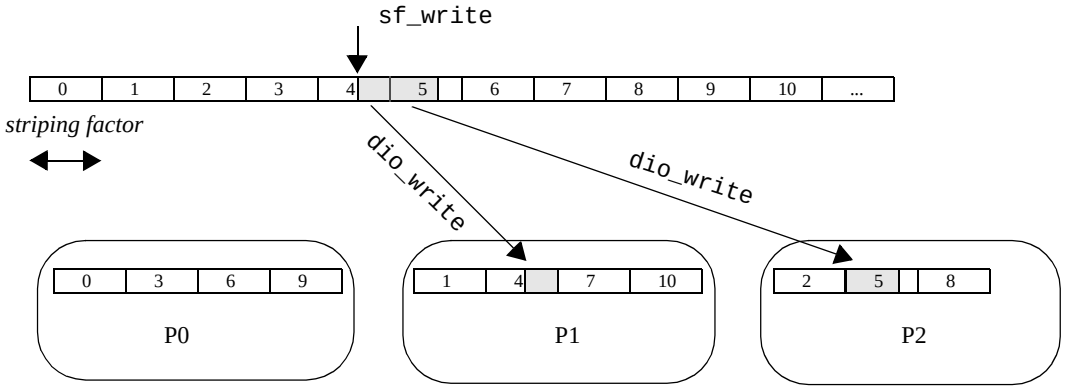

Figure 2: Example sf_write request decomposed into DIO operations on component files for a shared file implemented on three processors with local disks

- Shared Files support files larger than 2 GB in Fortran API.
- Shared Files read and write operations are nonblocking (a feature not yet available in Fortran-77/90) and contain an offset argument rather than a separate seek operation.

The SF 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. 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.

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 and the result undefined.

As we explain below, our DIO-based implementation can achieve significantly better performance than PIOFS (even with our well equipped IBM SP I/O configuration).

The Shared Files library is implemented by striping a parallel logical file across multiple physical files (component files) located on all disks available on the computing nodes on which the parallel application is running, see for Figure 2. It uses Distant I/O to perform remote read/write operations. Based on an input from the user for the "typical request size," SF determines a value of the striping factor. If this (optional) information is not available, an empirically determined value (e.g., 32 kB) is used.

The DIO implementation of Shared Files exploits the properties of this I/O model to avoid updating the size of component files when the application writes beyond the logical end-of-file boundary in the shared file. The size of a component file is updated when only read or write operation on that file is performed.

The original Shared Files implementation used the ELIO (Elementary IO) device library [6,7] as its portability layer. We added DIO as another device library compatible at run time with ELIO. This allows multiple filesystem implementations (e.g., PIOFS and local JFS files) in the same application and enables one to dynamically select, at run time, which implementation to use. The library uses the path specified in the metafile name to determine whether it points to a filesystem shared by all the processors (like PIOFS); if it does not, SF selects a DIO-based implementation for this particular shared file.

4 Performance

We measured the performance of DIO by writing and reading 1 GB (eight times the amount of main memory on a processor, to avoid caching effects) of data from both a local and a remote file. The request size varied from 4096 to 32,768 bytes. We tested two disk configurations on the 512-node IBM SP at Pacific Northwest National Laboratory (PNNL). In one configuration, the local JFS filesystem was

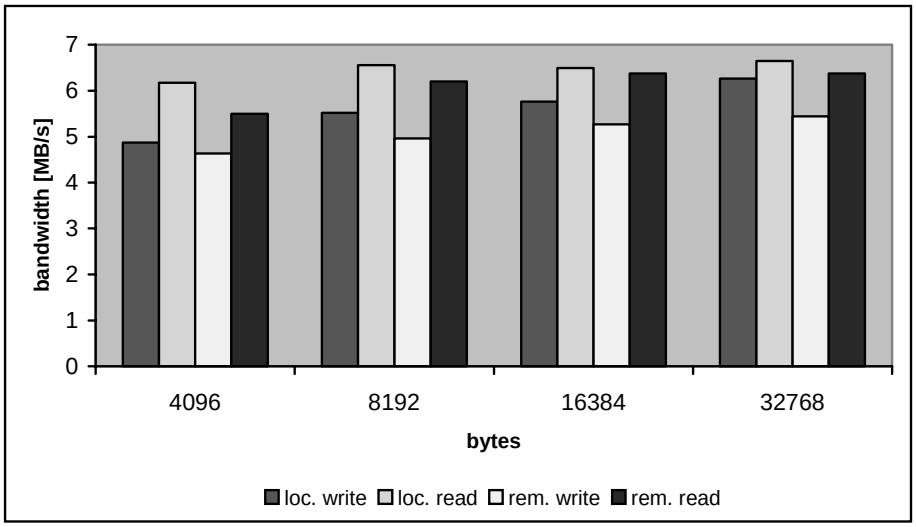

Figure 3: Performance of DIO *read* and *write* in access to local and remote files on the IBM SP (disks not striped)

mounted on a single SCSI disk; in the second configuration, it was striped on two identical SCSI disks (bandwidth optimization).

Figures 3 and 4 show that the I/O rates for remote requests approach local rates as the request size increases. The relatively small differences between local and remote I/O can be attributed to the efficiency of LAPI. (The bandwidth in LAPI_Put (used in dio_read) grows much faster with increased message size than in the MPI point-to-point message-passing [13] and, for example, achieves more than 70 MB/s bandwidth for messages as short as 4096-bytes.) However, the overhead associated with Active Messages processing (including the handler execution by a dedicated thread scheduled by AIX) contributes to the larger difference in performance for smaller requests.

5 Application

The multireference configuration interaction (MRCI) method is widely used in computational chemistry to obtain accurate predictions of properties of chemical systems. Details of the algorithms and methodology used in the COLUMBUS MRCI code are given elsewhere [11]. From the mathematical perspective, the program solves the eigenvalue problem for a very large sparse symmetric Hamilton matrix by using the standard Davidson method. A set of expansion vectors is used to project the original eigenvalue problem to a subspace eigenvalue problem that has a much smaller dimension. Because of the very high accuracy of the MRCI method, even for a very small molecule, the matrix becomes very large. To date, the state-of-the-art calculations in this area have been for 100 to 200 million of the configuration state functions (Hamilton matrix dimension). Despite using sparsity techniques and highly effective compression scheme to reduce the storage requirements such large calculations cannot be performed in-core.

To address memory limitations, COLUMBUS designers adopted a disk-based approach using the SF library [7]. This application is using shared memory programming model with data located in distributed system memory and in secondary storage The calculations are performed in an MIMD task-parallel fashion driven by data-dependent dynamic load balancing. The program allocates all possible distributed main memory for a one-dimensional global array [15] and then creates an SF shared file to store data that does not fit in the distributed main memory. 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 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 average I/O request size in this program is approximately 30 KB. Due to the algorithm properties and the amount of available in-core memory, the request size could not be increased.

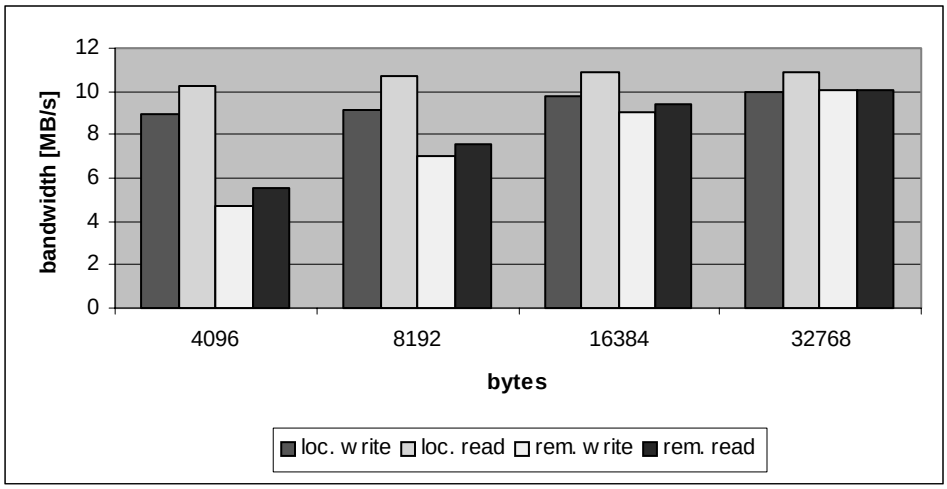

Figure 4: Performance of DIO *read* and *write* in access to local and remote files on the IBM SP (two disks striped)

We used both PIOFS and DIO implementations of Shared Files to solve the largest MRCI problem ever attempted, represented by a matrix of dimension 1.3 billion (1,295,937,374). The calculations were performed on 128 processors of the IBM SP at PNNL. We report timing results for two execution environments:

- SF-PIOFS (44 servers with 4 SSA disks each, which appears to be the largest PIOFS configuration available at that time anywhere) and MPL communication and
- SF-DIO on top of local disks (not striped) and the LAPI Active Messages library.

The I/O-related statistics are given (per iteration) in Table 1.

Table 1: I/O performance in COLUMBUS using Shared Files library on top of PIOFS and DIO

	Data Volume	I/O time	Bandwidth per CPU
PIOFS	900.963GB	906297.07s	0.994MB/s
DIO	900.957GB	235708.32s	3.823MB/s

The first column contains the amount of data read from secondary storage. It is slightly different for the two versions because of the dynamic nature of the load balancing and the algorithm properties. The second column indicates the total time wall-clock time spent reading data, summed over all processors. We do not report write rates because the I/O buffering layer in AIX makes them harder to measure reliably in a context of a complex application such as COLUMBUS rather than in a synthetic benchmark. The last column gives the average bandwidth rate per I/O request and per processor. These results show that the DIO-based implementation of SF outperforms PIOFS by a factor of four in this particular application. The SF-DIO read rates measured in COLUMBUS are only 25% lower than those measured by our microbenchmark for DIO alone (see Figure 3). This small difference is due to the overhead of the SF layer (striping) and some degree of contention when accessing data.

The overall execution time of COLUMBUS is 60% shorter for the DIO-based implementation than for PIOFS. The improvement can be contributed to increased efficiency of I/O and faster interprocessor communication. The original implementation of this application used the IBM MPL communication library and PIOFS. Although LAPI is faster (and incompatible) with MPL, this I/O-bound application is spending only 10% time on the interprocessor communication. Therefore, the increase of the overall performance than can be contributed to the improvement in I/O is estimated to be at least 50%.

6 Conclusions

We have proposed a new approach to parallel I/O in cluster systems, called Distant I/O. This new model avoids scalability problems and software overhead associated with client-server implementations of parallel filesystems, and supports the construction of scalable secondary storage systems based on attached disks. DIO provides one-sided and asynchronous access to disks attached to remote processors: in

effect, it extends the one-sided communication model to secondary storage. We implemented this model using active messages and Posix asynchronous I/O on the IBM SP. Our implementation approach is directly applicable to clusters of workstations on which various flavors of active-message–style communication facilities are available. We have used our DIO library both to construct applications directly and to implement higher-level libraries. In particular, we have constructed a DIO-based implementation of the Shared Files library by striping parallel-shared files on local disks attached the IBM SP nodes. DIO performance is excellent when measured by a microbenchmark and by a large computational chemistry application.

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. This work was performed under contract W-31-109-Eng-38 (Argonne National Laboratory) and under contract DE-AC06-76RLO 1830 (Pacific Northwest National Laboratory). This research was performed in part using 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] 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, November 1994.

[2] Compaq Computer Corp., Intel Corp., Microsoft Corp., Virtual Interface Architecture Specification, Dec. 16, 1997.

[3] S. A.Moyer and V. S. Sunderam, PIOUS: A scalable I/O system for distributed computing environments. In *Proc. Scalable High-Performance Computing Conf.*, 1994.

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

[5] 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.

[6] 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.

[7] J. Nieplocha I. Foster, R.A. Kendall, ChemIO: High-performance parallel I/O for computational chemistry applications, *Int. J. Supercomp. Apps. High Perf. Comp.* vol. 12, no. 3, 1998.

[8] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. *In Proc. Supercomputing'95*, 1995.

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

[10] J. Nieplocha, I. Foster, R. Kendall, ChemIO, http://www.emsl.pnl.gov:2080/parsoft/chemio.

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

[12] MPI Forum, MPI-2: Extensions to Message Passing Interface, University of Tennessee, July 18, 1997.

[13] 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. *Proc. 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing IPPS'98*, pages 260-266, 1998.

[14] C. C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-latency communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96*, November 1996.

[15] 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.