Non-collective Parallel I/O for Global Address Space Programming Models

Sriram Krishnamoorthy[†][*], Juan Piernas Canovas[‡], Vinod Tipparaju[‡],
Jarek Nieplocha[‡], P. Sadayappan[†]

[†] Dept. of Computer Science and Engineering, The Ohio State University
[‡] Pacific Northwest National Laboratory

Abstract

Achieving high performance for out-of-core applications typically involves explicit management of the movement of data between the disk and the physical memory. We are developing a programming environment in which the different levels of the memory hierarchy are handled efficiently in a unified transparent framework. In this paper, we present our experiences with implementing efficient non-collective I/O (GPC-IO) as part of this framework. A generalized mechanism (GPC) to invoke procedures on a remote node is implemented, which is then extended to handle non-collective I/O. We consider alternative approaches that can be employed in implementing this functionality. The approaches are evaluated using a representative computation from quantum chemistry. The results demonstrate that GPC-IO achieves better absolute execution times, strong-scaling, and weak-scaling than the alternatives considered.

1 Introduction

Out-of-core computations operate on data too large to fit in the collective physical memories of a parallel system. Typical approaches to out-of-core programs in a parallel system employ a collective approach, in which all the processes collectively perform disk I/O. This enables efficient disk I/O and communication schedules, such as optimized collective MPI I/O routines [4] and Disk Resident Arrays [9], to be employed to greatly reduce the disk I/O overhead and improve its scalability. Although this approach is very efficient, it requires the programmer to restructure the computation to identify and create points in the execution at which large aggregate amounts of data can be moved between disk and main memory. This can be challenging and tedious, especially for computations in which the disk I/O and computation patterns cannot be determined in advance.

[*]corresponding author: krishnsr@cse.ohio-state.edu

We are interested in dynamically structured computations that employ dynamic computation partitioning and disk I/O requirements. An in-memory computation with such characteristics can be efficiently implemented by exploiting support for *remote memory access* provided by communication libraries such as Aggregate Remote Memory Copy Interface (ARMCI) [15]. Using this approach, each process can access any block of data without synchronizing with other processes.

We are developing a programming framework, *extended global arrays* (XGA) in which the different levels of the memory hierarchy are handled in a unified transparent framework. In this framework, the data is assumed to be available in a global address space that can be accessed by any process without synchronization. A program written in this framework is automatically translated to be an in-memory or out-of-core computation as necessary. An integral part of this framework is the ability to non-collectively access data in secondary storage.

In this paper, we present our experiences in developing efficient non-collective disk I/O mechanisms for dynamically structured parallel applications. We investigate various approaches to providing non-collective access to out-of-core global data sets, in the same spirit as remote memory access. We are interested in data distributed amongst the local disks attached the processors in a cluster. While parallel systems attached to clusters can provide support for large secondary storage and efficient non-collective access to arbitrary data, they are typically shared with other applications in a cluster environment reducing the available throughput. In addition, a parallel file system has limited scalability, while the bandwidth available from local disks increases with the number of compute nodes available at the application's disposal.

Several applications can benefit from such a non-collective disk I/O functionality. We discuss some potential applications in Section 2. The extended global arrays framework that motivated our work is described in Section 3. Some alternative approaches to providing non-collective I/O support are considered in Section 4. A generalized mechanism (GPC) to invoke procedures on a remote node are implemented, which is then extended to handle non-collective I/O (GPC-IO). The implementation of GPC-IO is detailed in Section 5. The approaches are evaluated using a candidate computation, discussed in Section 6, from the quantum chemistry domain. The experimental results are presented in Section 7. Section 8 concludes the paper.

Note that this work does not provide generalized file system support and is not intended to replace the functionality provided by parallel files systems. Our solution to the problem focuses on providing low-level solution to light-weight management of out-of-core data sets on local disks.

2 Applications

In this section, we present some candidate scenarios in which support for efficient non-collective I/O can be beneficial.

Shared Files Shared Files [11] provides a logical file striped across the local disks of processors. While similar to parallel file systems in the support provided, the implementation utilizes system specific support on the IBM SP [16] to avoid daemons or server processes on the remote node. In addition, unlike typical parallel file systems, the user needs to simply link with the library to utilize the mechanisms provided, avoiding complex installations. The library was used to efficiently implement a large scale multi-reference configuration interaction (MRCI) calculation [11]. A portable yet efficient implementation of Shared Files requires efficient support for manipulating data on non-local disks.

Disk Resident Arrays Dist Resident Arrays (DRA) [9] provides a global multi-dimensional view of data on a parallel file system or on the local disks of nodes in a cluster. Collective operations are provided to move arbitrary multi-dimensional regions of data between global memory and secondary storage. Non-collective I/O support for disk resident arrays can extend the ease of out-of-core programming enabled by DRA, while opening up additional avenues of optimization. For example, when subsets of computation need access to disks, process groups can be created. The disk bandwidth available can be allocated to the process groups independent of the allocation of the computation. This would enable both finer control over disk resource management and better utilization by utilizing all available disks. Note that this can be done with a greater level of control than fixed striping of all files across all disks. Non-collective I/O also supports on-demand disk I/O, allowing data to be read just when required and written immediately after production. This can enable efficient memory management by eliminating buffers required to read all required data and write all produced data at global synchronization points.

Checkpointing Since a node failure also cuts off access to the disk on that node, checkpointing a computation requires state on a node needs to be written to disk on other processes. This is typically accomplished by a global barrier that enables the checkpointing of the global state. The increasing number of nodes modern clusters has resulted in applications partitioning the computation into logical sections each of which maintain relatively independent computation state. Such applications can benefit from providing capabilities for a process group to non-collectively access and checkpoint their state on a set of remote disks on another process group.

3 Motivation: Extended Global Arrays

Our primary motivation is the development of abstractions that integrates the three layers of the memory hierarchy – distributed main memory, shared memory on the SMP node of a cluster, and secondary storage – under a single programming interface.

3.1 Global Arrays

The Global Arrays toolkit presents to the application developer a distributed data structure as a single object and allows access as if it resided in shared memory. These features help the developer raise the level of composition and increase code reuse. A higher level of composition reduces the amount of code that must be written and enables scientists to program in terms of physically meaningful concepts rather than low-level manipulation of distributed data and explicit communication. Thus, it makes scientists more productive and permits more time to be spent optimizing performance-critical algorithms and application kernels. GA programming model includes as a subset message passing; in particular, the programmer can use full MPI functionality on both GA and non-GA data. The library can be used in C, C++, Fortran 77, Fortran 90 and Python programs. GA implements a shared-memory programming model in which data locality is managed by the programmer through explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to distributed shared-memory (DSM) models that provide an explicit acquire/release protocol. However, GA acknowledges that remote data is slower to access than is local data and therefore allows data locality to be explicitly specified and hence managed. Another advantage is that GA, by optimizing and moving only the data requested by the user, avoids issues such as false sharing or redundant data transfers present in some DSM solutions. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference. The GA toolkit provides extensive support for controlling array distribution and accessing locality information. Both task-parallel and data-parallel programming styles are possible. Task parallelism is supported through the one-sided (non-collective) copy operations that transfer data between global memory (distributed/shared array) and local memory. In addition, each process is able to access directly data held in a section of a global array that is logically assigned to that process. Atomic operations are provided that can be used to implement synchronization and ensure correctness of updates of overlapping array sections. The data parallel computing model is supported through the set of collectively called functions that operate on either entire arrays or sections of global arrays. The set includes BLAS-like operations interfaces to the parallel linear algebra libraries such as Scalapack as well as the TAO optimization toolkit [3].

3.2 Disk Resident Arrays

The disk resident arrays (DRA) [9] model extends the GA model to another level in the storage hierarchy, namely, secondary storage [10]. It introduces the concept of a disk resident array - a disk-based representation of an array. It 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.

3.3 SMP Arrays

So-called SMP Arrays (SA) can be used as a shared memory cache for latency sensitive distributed arrays in cluster environments based on collection of Symmetric Multiprocessor (SMP) nodes. Due to its cost effectiveness, SMP systems are used as building blocks for both commodity clusters as well as custom architectures (e.g., IBM SP, SGI Altix, NEC SX, Cray X1). SA arrays resemble global arrays except their scope is limited to an SMP node rather than entire parallel job running on a cluster. SA are related to the mirrored arrays, that were initially introduced as an extension to Global Array model in context of wide-area-network grid computing environments [12–14] and recently proposed for reducing communication overhead on cluster [17]. In the latter context, shared memory mirroring is used to cache entire global arrays on every SMP node. The arrays are replicated across cluster nodes and distributed within each node. The goal is to take performance advantage of the shared memory, which constitutes the fastest interprocessor communication protocol, and use it as replacement for more expensive network communication. In the mirrored approach, the user is responsible for managing consistency of the cached data and collective operations on arrays are globally synchronized. The SA arrays do not involve global synchronization in collective operations and are created and managed independently on each SMP node.

3.4 Integrated Programming Framework

The evolution of programming models is driven by the fundamental trade-offs between high productivity and performance requirements in context of evolving scalable architectures. On one hand, high productivity demands high-level of abstractions that insulate the programmer from specificity of the underlying hardware details and allow describe the underlying mathematical model in terms of collection of algorithms and appropriate data structures. However, achieving high performance and scalability is difficult if the essential characteristics of the hardware, in particular the memory hierarchy, are ignored. Intelligent and automated management of data movement is a fundamental and unifying theme for the Extended Global Array interface we are developing. The goal is to have a single interface for managing data and high level representation of the mathematical algorithms operating on multidimensional arrays while the details on the underlying data movement between secondary storage, distributed memory, shared memory, and local memory are handled by the XGA implementation. XGA attempts to address this problem while relying on three elements:

- Compiler analysis and code transformation
- Performance model for GA, SA, DRA operations
- Information on resource availability and configuration (disk space, memory, processor affinity).

The basic idea is to translate XGA programs into SA/GA/DRA code while orchestrating data movement, caching, and redistribution so that the performance is maximized while satisfying the constraints on the available resources. XGA would allow from a single source to generate in-core and out-of-core codes while reducing the programmer effort and maintenance costs. Data locality is improved by incorporating locality in the data structure to minimize data movement costs [8].

An XGA programming model that is as convenient to program as a shared memory system cannot require all data movement operations to collective in nature. On the other hand, automatic translation of non-collective data movement operations to collective operations for arbitrary XGA programs is a challenging task. Thus support for efficient non-collective operations on out-of-core data accessed by XGA programs is critical to the widespread applicability of the XGA programming model.

4 Non-collective I/O: Approaches

In this section, we discuss two possible approaches to implementing non-collective I/O.

4.1 Replication-Based Approach

In this approach, all processes create files to hold all the blocks. The input arrays are replicated. Each process can then access the blocks of the input arrays without synchronizing with other processes. Note that different blocks of the output array are computed by distinct processes, requiring a replication of the result before it can subsequently be used as an input. No communication cost is incurred during the computation. But the replication cost is inherently non-scalable, and incurs an increasing fraction of the total computation time with an increase in the number of processes.

We observe another limitation that inhibits scalability. In a collective I/O operation, each process reads/writes data from its local disk. This results in a block residing in the operating system buffer cache of the process reading/writing that block, resulting in faster response times on subsequent accesses to the same block of data. The collective buffer cache in the system increases with increase in the number of processors. Since each block of data is accessed from the disk by only one process, the number of distinct blocks of data stored in the buffer cache increases linearly with the number of processors. In a replicated computation, the same block of data can end up residing in the buffer cache of multiple processors. This results in an inefficient utilization of the system buffer cache, inhibiting scalability.

4.2 Parallel File System Based Approach

Alternatively, a parallel file system, such as Lustre [1] or PVFS [2, 7], can be mounted on the local disks of the processes. Each process creates a distinct file to store the blocks of data alloted to it. Since these files are created on the mounted parallel system, all processes can access the files. This enables non-collective access to any block of data without explicit replication.

But the application of a parallel file system comes with some disadvantages. The dynamic installation of a parallel file system on a subset of nodes allocated to the application is not always straightforward. In addition, the limited user interface restricts explicit control of data distribution. The files can potentially be striped across the disks. This is useful for sequential programs that would like to implicitly take advantage of the parallelism while handling disk I/O movement. In addition, applications that cannot benefit from specialized handling of data distribution can be satisfied with striped files. We are interested in a complementary scenario in which the programmer handles the parallelism and hence data distribution, while the disk I/O is potentially implicit. In general, striping results in multiple requests for proportionally smaller data units. This results in lesser locality of access at the disk level, interfering with disk prefetch optimization and increasing disk seek times. We further explain our observations in Section 7.

Note that non-collective MPI I/O [4] routines utilize a parallel file system underneath when operate on data on local disks. Hence, they would exhibit the characteristics of the underlying parallel file system.

5 GPC-IO: Design and Implementation

An earlier implementation [16] of non-collective I/O utilized LAPI [18], a commercial Active Messages [6] implementation on IBM SP. While it demonstrated good performance, it was limited in its applicability. We are interested in a portable implementation that achieves high performance on a variety of platforms. In addition, the implementation strategy should not require an extensive installation procedure to draw upon its capabilities. For example, Lustre file system [1] provides extensive support, through a rich set of primitives, for I/O on disks attached to the nodes in a cluster. While I/O nodes that contain the file system are typically separate from the nodes performing the computation, it can be configured to operate on the subset of the cluster nodes that are allocated to a particular computation. This requires configuring Lustre at the start of each job.

We have developed a non-collective I/O mechanism, GPC-IO, within the GA/DRA framework. The non-collective I/O functionality was layered on top of *Global Procedure Calls* (GPC). Global procedure calls are invocations of pre-registered procedures at a remote node. A global procedure handle is obtained by collective registration of a procedure. The handle can then be used to invoke the registered procedure at a remote node.

The design of the global procedure call mechanism was guided by the following objectives:

Extensibility The implementation should be usable in a variety of contexts. In particular, it should allow the manipulation of data at a remote node before returning a result. This can enable optimizing functions such as non-collective reductions. A GPC call can be used to reduce the data (such as finding the sum, minimum, maximum, etc.) at the remote node and only communicate the reduced result, lowering the communication volume.

Semantic simplicity The implementation, while being powerful, should not complicate programming of such applications. For example, allowing the global procedure call to access the remote process context would require the programmer to handle data sharing and mutual exclusion issue between the GPC and the main flow of the control at the remote process. This would in turn require the user to handle all the complexities of a full-blown multi-threaded programming model. On the other hand, denying access to data at the remote node will preclude optimization such as remote reductions. A parallel programming model based on remote memory access requires the programmer to handle concurrent access to portions of global data. We employ similar semantics – a GPC can access portion of the global data that is resident on the remote node. This ensures a powerful mechanism without making it any more complex than remote memory access.

Low overhead A GPC call cost should be as close to that of a remote access operation. This would help in its utilization in a wide variety of scenarios.

Implementation portability We implemented the GPC mechanism within the Global Arrays framework. ARMCI provides a rich set of highly optimized communication primitives that are portable across a wide variety of platforms. We leveraged this implementation to provide the global procedure call functionality on a variety of platforms.

5.1 Illustration

The implementation of global procedure calls is illustrated using the call sequence involved in a typical GPC invocation.

Figure 1 shows the data structures and data movements in a GPC call. Each of the steps involved are denoted by circled numbers. The process initiating a GPC call passes the header and data for the GPC request that are carried through to the GPC callback function on the remote process that will be executing this GPC. This is indicated in the figure as step 1. Along with the request header and data, the process also passes pointers to the response header and response data if a response is expected. For example, a remote disk write request will contain data that needs to be written. A remote read request will have empty request information and a buffer to copy the response data upon the completion of the operation. This request is sent to the remote process (step 2) and upon arrival at the remote process, it is processed by the data server process (show in the Figure as step 3). The data server executes the previously registered callback function and passes the request header and data on to the callback function. This is shown as step 4 in the figure. The callback function is executed and the header and data from the response are sent back by the remote process to the process that initiated the GPC (steps 5 and 6). The process initiating the GPC call has 2 options. It can either block on the GPC call or issue it and check for completion using the GPC non-blocking handle.

The non-collective read and write operations are implemented as lightweight wrappers over the GPC calls that utilize appropriate callback functions that read and write from the local disk at the node on which the they are invoked.

6 Tensor Contractions in Quantum Chemistry

We evaluated our approach on tensor contractions encountered in certain quantum chemistry calculations. In this section, we describe the implementation

Ab Initio quantum chemistry models such as Coupled Cluster methods [5] involves computations expressible as a sequence of tensor contractions. Each tensor contraction expression is comprised of a collection of multi-dimensional summations of products of several block-sparse input arrays. Consider the

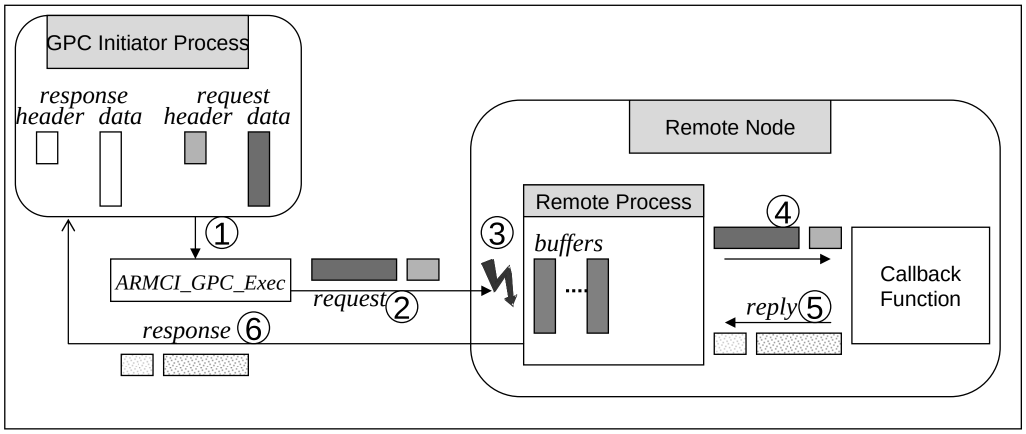


Figure 1: Illustration of non-collective operation

following tensor contraction from the domain of quantum chemistry:

$$p1, p2, p3, p4 : O$$
$$h1 : V$$
$$i0[p1, p2] \mathrel{+}= \; t[p1, p3, p4, h1] * i1[p3, p4, h1, p3]$$

where indices $p3$, $p4$, and $h1$ are contracted out. Here O is the number of occupied orbitals, and V is the number of virtual orbitals. O and V are divided into segments. This segmenting of the dimensions forms a cartesian grid that divides the multi-dimensional array into blocks. An operation on the indices of the segments that form a block determines if that block is non-zero.

Despite being a variant of matrix-matrix multiply, the block-sparsity in tensor contractions leads to irregular data access patterns that are not easily tractable. An efficient implementation of the tensor contractions uses a data structure in which the non-zero regions of the tensors are blocked. This enables the use of efficient BLAS kernels to optimize the sequential computation of a block, while enabling good load-balancing by partitioning the work in terms of the blocks.

The sizes of O and V are such that the arrays are too large into fit into the collective physical memory of a parallel system. The arrays are usually stored on the local disks attached to the compute nodes in a cluster, to achieve scalable I/O.

Consider the tensor contraction in the above example. The non-zero blocks of tensor $i0$ are computed by the processors, with each block being computed by a separate processor. Consider a partitioning of the input arrays t and $i1$ amongst the local disks attached to the processor. A block of the input arrays is typically involved in the computation of multiple blocks of the output array. Due to this data reuse relationship in the computation, it is not feasible to achieve a load-balanced partitioning of the tensor contraction such that all

k	#procs			
	2	4	8	16
1	500	558	560	809
2	1045	1279	1228	1990
4	2246	2455	2511	2721

Table 1: Cost, in seconds, of replicating the input arrays

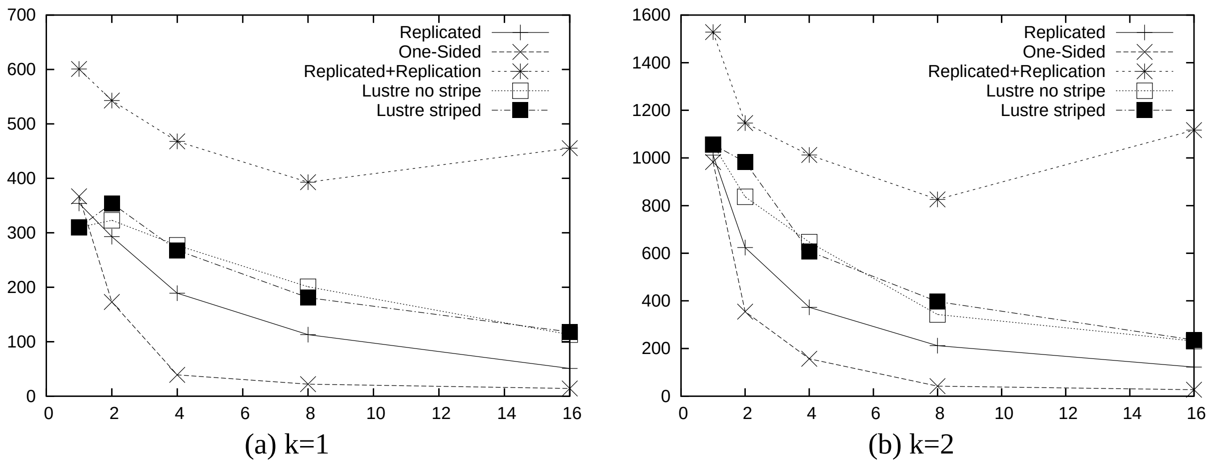


Figure 2: Execution times, in seconds, for the various approaches for (a) k=1 and (b) k=2

the input data blocks required for the computation of blocks assigned to a processor can be located in that processor. In addition, The variation in the block sizes, together with the variation in the data movement costs incurred, makes a simple static partitioning scheme unattractive. A dynamic partitioning scheme is typically employed to better balance the load and ensure that all processors are actively contributing to useful work.

The dynamic nature of the computations also precludes the use of collective operations to read and write bricks, which would lead to excessive load-imbalance and severe performance degradation. The different approaches to non-collective I/O, including GPC-IO, were implemented to access the blocks of the input tensors dynamically without synchronization.

7 Experimental Results

We evaluated the three approaches to non-collective I/O – file replication, employing a parallel file system, and GPC-IO. The execution times for the tensor expression described in Section 6 were measured. The

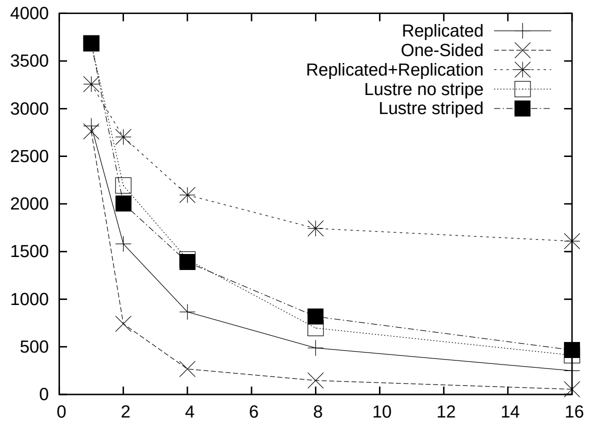
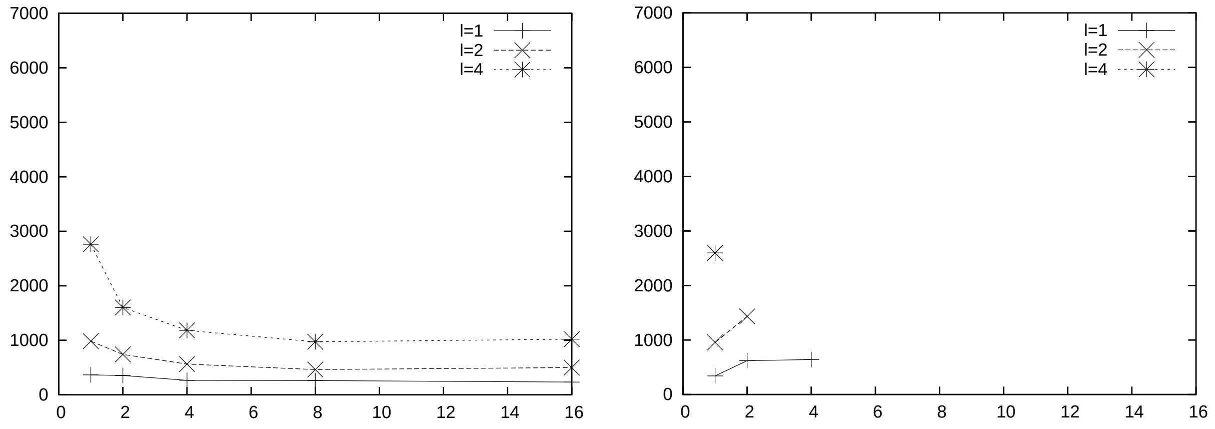
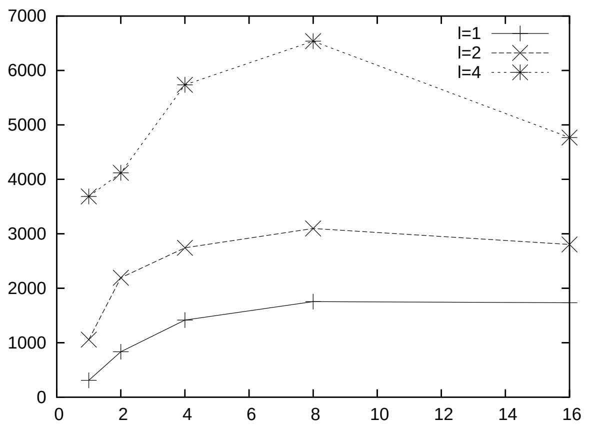


Figure 3: Execution times, in seconds, for the various approaches for k=4

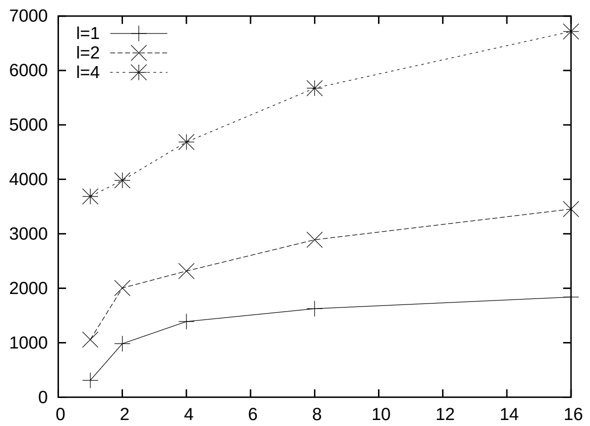


(a) GPC-IO (b) Replicated

Figure 4: Weak scaling of the GPC-IO and replication-based approaches for different starting problem sizes. A linear scaling would correspond to a perfect horizontal line.



(a) Lustre no striping



(b) Lustre striped

Figure 5: Weak scaling of the approaches based on Lustre for different starting problem sizes. A linear scaling would correspond to a perfect horizontal line.

occupied orbitals (O), was set to consist of four segments – 100, 60, 40, and 30, respectively. The virtual orbitals segments (V) were set to be a multiple of the occupied orbital segments ($V = k \times O$).

The schemes were evaluated on the Colony2a cluster at the Pacific Northwest National Laboratory. Each node in the cluster is a dual 1 GHz Itanium-2 system, with 6GB physical memory, 80GB hard drive and a Myrinet interconnection network. All the experiments were performed by utilizing one processor per node.

We measured the execution times for the replication-based approach with and without considering the cost of file replication. This comparison should bring out the difference in performance of the core algorithm without considering the replication costs, which could potentially be amortized when input arrays are used in more than one contraction. Table 1 shows the cost of replicating the input arrays t and $i1$ for different problem sizes, determined by k, and number of processors. As noted in the algorithm, the results demonstrate that the replication operation is not scalable, and the replication cost increases with the number of processors. Note that the implementation does not overlap disk I/O with communication, which can reduce the replication cost by up to a factor of two. We define the "optimal" replication cost to be half the cost shown in Table 1. To ensure fairness, all comparisons will be made using the "optimal" replication costs instead of the actual ones.

In the parallel file system experiments, every compute node is also a storage node (OST) of a Lustre 1.6beta5 file system. There is also an additional node, other than the nodes used to perform the computation, which is used as meta-data server (MDS) only. We evaluate the tensor contractions without striping and with

128KB striping across the OSTs. All the compute nodes mount the parallel file system, which stores the files required by the application. Each process creates a distinct file to store the data blocks alloted to it. Since these files are created on the Lustre file system, all processes can access any file.

Figure 2 and Figure 3 show the execution times of the three schemes for different values of k. GPC-IO is consistently better than the other alternative approaches. The observed super-linear speed-ups can be attributed to the efficient utilization of the collective system buffer caches and the increase in the available disk bandwidth. Note that the implementation can be improved further by overlapping computation with disk I/O and employing a client-side cache to reduce data movement costs.

The replication-based approach performs worse than GPC-IO, even without taking the cost of replication into account. The factor of degradation as compared to GPC-IO reaches more than 5. The factor of degradation when including the replication cost is more than 10. The degradation is due to the poorer exploitation of locality. A brick cached in the disk cache of a processor can serve all incoming requests for that brick without going to disk. In the GPC-IO scheme, disk I/O for any given data brick happens in at most one processor, with other processors obtaining that brick through communication. This results in each brick being in at most one disk cache. Thus increase in the number of processors increases exploitation of reuse in the distributed buffer caches.

The Lustre file system based scheme does not scale well either. The speed-up improves with larger problem sizes and larger numbers of processors, reaching 7.9 for $k = 4$ for 16 processors.

Lustre implements client-side caching, which has an effect similar to replication in reducing the communication costs. On the other hand, the limited server-side caching in Lustre results multiple I/O requests to the same block of data from different processes results in redundant I/O. In addition, the approach suffers from a lack of finer control over data distribution. In the non-striped evaluation, each file is fully created in one of the OSTs, not necessarily in the local disk of the creating process. In the striped evaluation, the stripe size is fixed for the entire file and can result in a block being split into multiple smaller pieces. This results in an increase in the effective time spent in disk seeks on all the local disks, thus reducing the overall bandwidth available. An increase in the number of processors results in a decrease in the number of tasks, and the data requested, by each process. For larger numbers or processors, the reduction in the total data requested by each process coupled with aggressive client-side caching results in a reduction in the disk bandwidth penalty incurred.

Figure 4 shows the weak-scaling measurements of the contraction for larger problem sizes for GPC-IO and replication-based schemes. Figure 5 shows the weak-scaling measurements for the Lustre-based approach. Three problem size classes are considered – l=1,2, and 4. The actual problem size for a given data point is determined as $k = l \times p$, where p is the number of processors. The graphs plot the number of processors along the x-axis and the execution time along the y-axis. Linear weak-scaling will be observed in the graphs as a horizontal line parallel to the x-axis. All graphs are plotted on the same scale to ease comparison across graphs.

We observe that GPC-IO achieves super-linear weak scaling and better execution times than other approaches. The increase in disk bandwidth and computation capacity with increase in the number of nodes facilitates linear weak scaling. The increase in the physical memory available supports better exploitation of locality in the computation, resulting in the observed super-linear weak-scaling. When no further locality can be exploited in the computation we observe linear scaling.

For the replication-based approach, the cost of replication is not included. Larger problem sizes could not be evaluated for the replication-based scheme, due to the limited local disk space (¡80GB) available on the cluster.

The Lustre-based approach suffers from severe performance degradation for larger processors due to limited control over data distribution and hence data movement patterns.

8 Conclusions

In this paper, we presented our experiences with implementing efficient non-collective I/O (GPC-IO) as part of this framework. A generalized mechanism (GPC) to invoke procedures on a remote node was implemented, which was then extended to handle non-collective I/O. We considered alternative approaches that could be employed in implementing this functionality. The approaches were evaluated using a representative computation from quantum chemistry. The results demonstrated that GPC-IO achieves better absolute execution times, strong-scaling, and weak-scaling than the alternatives considered.

References

[1] Lustre File System. http://www.lustre.org.

[2] Parallel Virtual File System. http://www.pvfs.org.

[3] Steve Benson, Lois Curfman McInnes, Jorge J. More, and Jason Sarich. TAO Users Manual. Technical Report ANL/MCS-TM-242-Revision 1.5, Argonne National Laboratory, 2003.

[4] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NAS, January 1995.

[5] T. Crawford and H. Schaefer III. An Introduction to Coupled Cluster Theory for Computational Chemists. In K. Lipkowitz and D. Boyd, editor, *Reviews in Computational Chemistry*, volume 14, pages 33–136. John Wiley & Sons, Ltd., 2000.

[6] David Culler, Kim Keeton, Lok Tim Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. The generic active message interface specification. Technical report, University of California, Berkeley, 1994.

[7] W.B. Ligon III and R. B. Ross. *Beowulf Cluster Computing with Linux*, chapter PVFS: Parallel Virtual File System, pages 391–430. MIT Press, 2001.

[8] Sriram Krishnamoorthy, Umit Catalyurek, Jarek Nieplocha, Atanas Rountev, and P. Sadayappan. Hypergraph partitioning for automatic memory hierarchy management. In *Proc. Supercomputing (SC 2006)*, November 2006.

[9] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-Of-Core Computations. In *Proc. 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204, 1996.

[10] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations. In *6th Symposium on the Frontiers of Massively Parallel Computing*, March 1996.

[11] J. Nieplocha, I. Foster, and R. Kendall. Chemio: High performance parallel i/o for computational chemistry applications. *Int. J. Supercomp. Apps. High Perf. Comp.*, 12(3), 1998.

[12] J. Nieplocha and R. J. Harrison. Shared memory NUMA programming on I-WAY. In *5th International Symposium on High Performance Distributed Computer (HPDC-5)*, pages 432–441, 1996.

[13] J. Nieplocha and R. J. Harrison. Shared-Memory Programming in Metacomputing Environments: The Global Array Approach. *J. Supercomputing*, 11:119–136, 1997.

[14] J. Nieplocha, R. J. Harrison, and I. Foster. Explicit management of memory hierarchy. In *Advances in High Performance Computing, NATO ASI Series 30*, pages 185–198. Kluwer Academic, 1996.

[15] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Workshop on Runtime Systems for Parallel Programming (RTSPP)*, 1999.

[16] Jarek Nieplocha, Ian Foster, and Holger Dachsel. Distant I/O: One-sided access to secondary storage on remote processors. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, 1998.

[17] B. Palmer, J. Nieplocha, , and E. Apra. Shared memory mirroring for reducing communication overhead on commodity networks. In *5th International Conference on Cluster Computing (CLUSTER 2003)*, 2003.

[18] 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. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, page 260, Washington, DC, USA, 1998. IEEE Computer Society.

Acknowledgments

We thank the National Science Foundation for the support of this research through grants 0121676 and 0403342, and the U.S. Department of Energy through award DE-AC05-00OR22725. We thank the Molecular Sciences Computing Facility (MSCF) at the Pacific Northwest National Laboratory (PNNL) for the use of their computing facilities.