A Pragmatic Approach to Improving the Large-scale Parallel I/O Performance of Scientific Applications.

Lonnie D. Crosby, R. Glenn Brook, Bhanu Rekepalli, Mikhail Sekachev, Aaron Vose, and Kwai Wong
National Institute for Computational Sciences, University of Tennessee at Knoxville

ABSTRACT: I/O performance in scientific applications is an often neglected area of concern during performance optimizations. However, various scientific applications have been identified which benefit from I/O improvements due to the volume of data or number of compute processes utilized. This work details the I/O patterns and data layouts of real scientific applications, discusses their impacts, and demonstrates pragmatic approaches to improve I/O performance.

KEYWORDS: MPI-IO, IO, Lustre, Application Performance

1 Introduction

Many factors affect the performance of scientific applications such as utilization of vector processing instructions, memory access patterns, cache utilization, and interprocess communication patterns. The application's interaction with the memory, processor, and interconnect of computing resources has long been a focus for improving performance. However, the application's interaction with the file system has recently become a topic of discussion. Some reasons for this renewed attention include the increasingly data-intensive nature of computation and analysis, the increasing size of computational resources, and the increased availability of shared supercomputing resources.

Various studies have been performed which illuminate the characteristics of file systems and their interaction with specific application patterns [1, 2]. However, in terms of many established scientific applications, these patterns are overly simplistic. Scientific applications, based in part on their field, constrain the applicability of I/O patterns and, in some cases, define particular data formats. These considerations affect the routes that can be utilized to optimize performance. The goal of this work is to demonstrate the connection between more theoretical studies of I/O performance and their application to the constraints and requirements of real scientific applications.

1.1 Parallel I/O Performance

The application's utilization of file I/O is not unlike other forms of data movement such as interprocess communication and memory access. However, file I/O requires the interaction between the memory, interconnect, and file system. While all these interactions are subject to limitations in bandwidth and latency, the file system interaction is by far the most expensive. Therefore, most I/O performance optimizations focus on the file system interaction.

Some guidelines include:

1. Limit the impact of latency by performing I/O in as few large chunks as possible.
2. Limit overhead by performing as few metadata operations (file open/close, stat, and seek) as possible.
3. Write data contiguously whenever possible.
4. Avoid contention on the file system.

A large portion of the I/O performance rests in the choice of output data format. This choice most directly affects the above guidelines; however, the data layout in memory and across processes (for parallel I/O) have a significant impact on performance. This impact on performance can be due to poor memory or interconnect usage patterns. However, the mechanism by which data in memory or across processes is mapped to the output data format is a significant contributor to performance. Although, the same data layouts are rarely utilized between the application (in memory and across processes) and the output. The best performance is seen when these data layouts are similar. Differing application and output data formats create constraints to I/O optimization. The methods utilized to satisfy these constraints may have significant performance implications.

All results presented in this paper were generated on Kraken, the Cray XT5 supercomputer located at the National Institute for Computational Sciences (NICS).

Kraken contains 9,408 compute nodes, each containing 2 hex-core AMD Istanbul processors, 16 GB of RAM, and a SeaStar 2+ interconnect [3]. Kraken utilizes a Lustre [4] parallel file system with 48 OSSs and 336 OSTs that is capable of delivering about 30 GB/s in peak performance. The file striping characteristics are presented with the appropriate results.

1.2 Applications

Three applications were chosen for I/O optimization based on a perceived need by the application developers and/or users. Although performance improvements may be seen by changing the output data format and/or the application's data structures, these improvements are impractical for many applications due to dependencies in computational modules or post-processing utilities. In this work, the output data format for each application is kept constant so that optimizations do not break backward compatibility. Additionally, existing data structures within the computational kernels of these applications are also kept constant. Only data structures used within the I/O portions of these applications are altered. Therefore, the I/O optimization of these three applications focuses on the mapping between application and output data.

The Parallel Interoperable Computational Mechanics Simulation System (PICMSS) is a fully parallel computational platform for solving various incompressible computational fluid dynamics (CFD) problems using finite-element based methods. This code is developed by a group of engineers at the University of Tennessee's CFD Laboratory.[5] This work applies PICMSS to a validation benchmark problem that models a three-dimensional flow in a thermally-driven cubic cavity.

The AWP-ODC [6] application is utilized to conduct the "M8" simulation, which models how the ground will shake in a magnitude 8.0 earthquake on the southern San Andreas Fault up to 2-Hz. This simulation models an area of approximately 125,000 square miles (328,050 square kilometers) and takes into account a depth of 50 miles (85 km) for an earthquake 6 minutes in duration. This software is developed by a multi-disciplinary team of researchers coordinated by Southern California Earthquake Center (SCEC) at the University of Southern California (USC).

A parallel implementation of the Basic Local Alignment Search Tool (BLAST)[7] has been developed by researchers at the University of Tennessee.[8] BLAST compares input nucleotide or protein sequences to sequences in a database and reports matches along with other information, such as the degree of similarity of matched regions within the sequence. The database used in this work is a recently obtained non-redundant protein database which contains 13,663,181 protein sequences. The total number of amino acids present in the 10.7 GB database is 4,688,826,815. The input query sequences were selected based on common searches used by biologists ranging from 50 to 5,000 amino acids.

Throughout the remainder of this work the application's name is not used. Instead, a designation of "Application 1 - 3" is used to identify these applications in no particular order. The output data format and relevant application data structures are discussed for each application. The original I/O pattern and the optimizations performed are discussed in terms of factors which affect performance. Finally, a quantification of performance is discussed which compares the original and optimized application versions.

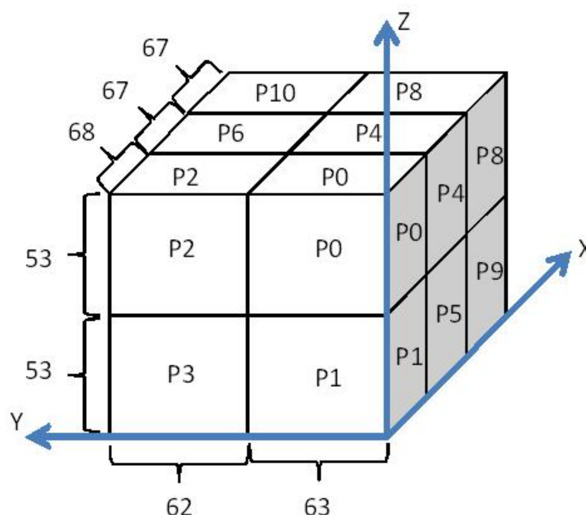

Figure 1: The domain decomposition of a 202x125x106 grid among 12 processes in a 3x2x2 grid. The process assignments are listed P0-P11 and the numbers in brackets detail the number of grid nodes along each direction for each process block.

2 Application 1

The computational grid consists of 10,125 x 5,000 x 1,060 nodes. However, every other point in each direction is skipped during I/O, resulting in an I/O grid of 5,062 x 2,500 x 530 nodes. Each node stores three variables. This grid is decomposed among 30,000 processes via a process grid of dimension 75x40x10. Nodes are assigned to processors as evenly as possible giving each process a local grid between 67x62x53 and 68x63x53 nodes. Figure 1 shows this decomposition for a grid of 202x125x106 nodes and a process grid of 3x2x2. This global grid is written in column-major ordering.

For each process, the local grid is stored in three column-major ordered arrays which correspond to each variable. This application has the ability to aggregate time steps in these arrays by simple concatenation (i.e. the data for time step 2 comes immediately after all data for time step 1). As a result, these arrays may contain data for multiple time steps.

Three output files are written which correspond to each of the three variables and the number of aggregate time steps. The global grid is ordered column-major for each time step. Multiple time steps are added by simple concatenation such that time step 2 comes immediately after all data for time step 1. Subsequent writes create a new set of files which contain additional time steps. MPI-IO [9, 10, 11] is utilized to write these files.

2.1 Optimization

Due to the output data format, data from each process is spread throughout the entire file. Data is spread within a single time step based on the process's location within the process grid and between each time step based on the number aggregated. The original mapping between the local data arrays and the output file format is controlled via a MPI derived data type created by the MPI_Type_create_hindexed call. Every element of this array is individually mapped to a location within the output file via an explicit offset. This allows a single collective write (MPI_File_write_all) to be used for each file. Although this utilization of a MPI derived data type is effective, the implementation does not allow for contiguous writes when possible. Each process should be able to write at least 67 array elements contiguously; however the mapping of individual elements does not give the MPI-IO library sufficient information to exploit this optimization. Additionally, the mapping of every array element causes problems with the use of explicit offsets for large data sets and for large amounts of time step aggregation.

After optimization, the mapping between the local data arrays and the output file is achieved via a MPI derived data type created by the MPI_Type_create_subarray call. This call provides the necessary information to the MPI-IO library to allow contiguous writes to be utilized when appropriate. Additionally, this call circumvents the use of explicit offsets which may cause problems with large data sets. This subarray data type is also defined with an extent equal to the global data set (i.e. a single time step) which reduces the overhead associated with its definition. This optimization can also be implemented with the previous h-indexed data type. A single collective write is still used to write each file.

2.2 Results

The benchmark case utilizes 30,000 compute cores run for 200 time steps. Each output file contains 20 time steps striped across 160 OSTs with a stripe size of 1MB. The I/O time is measured by timing each write and taking the maximum process time. Table 1 shows the I/O performance for the original and optimized case. Although performance varies, a consistent improvement in performance by a factor approaching two is observed. Given the amount of data written, this optimization reduces the I/O time by about 12 minutes for each write. Over 200 time steps, this savings approaches about 2 hours.

Table 1: Comparison between Original and Optimized I/O in Application 1

		Bandwidth (GB/s)	
Time Step	Data (TB)	Original	Optimized
20	1.46	1.05	2.03
40	1.46	1.06	1.66
60	1.46	1.40	2.16
80	1.46	1.16	1.94
100	1.46	1.27	2.30
120	1.46	1.55	2.06
140	1.46	0.98	1.70
160	1.46	1.30	2.65
180	1.46	0.87	3.34
200	1.46	1.13	2.99

3 Application 2

This application relies on task based parallelism in which each process obtains work (data) from a master process. This division of labor is hierarchical such that an application master process serves data to node-based master processes who serves this data to worker processes on each node. The hierarchy of application and node-level master processes performs the necessary inter-node communication.

Upon completion of a task, each worker sends the results in XML format to a writer process whose task is to write the output to disk. Each node has a single writer process who obtains output from all the worker processes on the node. All XML data is concatenated and compressed with zlib library [12]. Each node writes a single file which contains one or more of these compressed blocks of XML data. A 4-byte header is prepended to each block to identify the block and its length.

3.1 Optimization

The original implementation of this I/O scheme utilizes on-demand processing. Each task output is compressed and written to disk individually and immediately upon arrival by the writer process. This method, though effective at keeping the writer process busy, results in many small file writes. Additionally, the 4-byte header for each compressed block, is written after the block which requires additional file seeks.

To optimize the I/O, two buffers are set up to collect compressed and uncompressed XML task output. The first buffer collects and concatenates XML output from worker processes. Once this buffer is full, the data is compressed and placed in a second buffer. The second buffer collects compressed XML data and associated 4-byte headers. When this buffer is full, the data is written to the output file. This two-stage buffering minimizes the overhead connected with the data compression and the file write.

3.2 Results

The benchmark case utilizes 24,576 compute cores (2,048 nodes). Each node contains a single writer process that performs both data compression and I/O. Each compression and write operation performed by this process is timed and summed for the total execution time of the application. Table 2 shows the average compression or write time per writer process along with the average bandwidth for the original and optimized application code. Both the compression and write buffers are 768 MB. Each file is striped across one OST with a stripe size of 1 MB.

Table 2: Comparison between Original and Optimized I/O in Application 2

	Original	Optimized
Average Compression Time (s)	11.93	8.85
Std. Dev. (s)	0.79	0.46
Bandwidth (MB/s)	25.75	34.63
Average Write Time (s)	10.47	0.30
Std. Dev. (s)	8.36	1.06
Bandwidth (MB/s)	16.04	466.67

The efficiency of both compression and write operations are improved. An improvement of about 33% is observed for data compression. However, an improvement of about 29-fold is observed in the data write. Another effect of these optimizations is an increase in the regularity of I/O, which is expressed by a decrease in the standard deviation.

With these optimizations, the time spent performing I/O has been reduced by a factor of about 30. Given that an average compression ratio of about 1:7.5 is achieved in these benchmarks and the improved I/O rate, the uncompressed data may be able to be written to disk in less time than is required by the compression.

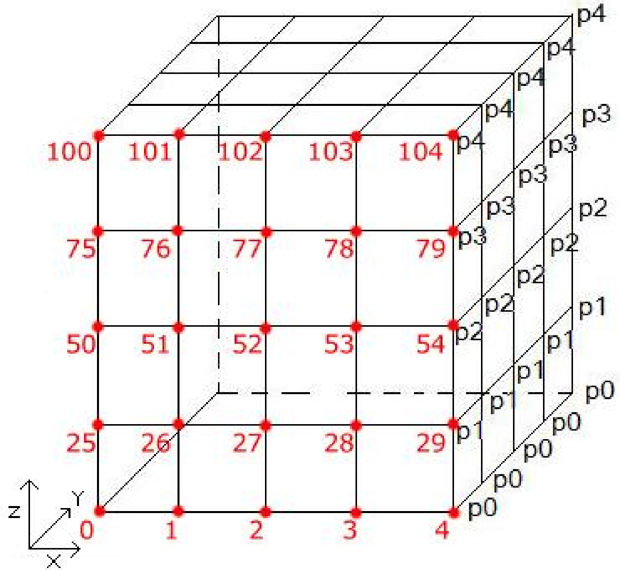

Figure 2: A 3-D 5x5x5 Cartesian grid decomposed in slabs along the Z axis among 5 processes (p0-p5). Node numbers are shown in red and process IDs are shown in black.

4 Application 3

A uniform 3-D Cartesian grid is utilized with a total of 256^3 nodes. Each node stores 6 variables. This grid is decomposed between the processes in slabs along the Z-axis. This is shown in Figure 2 for a 5x5x5 grid decomposed between 5 processors. However, this application decomposes its 256^3 node grid between 3,000 processes. In this case, each process is only assigned a portion of a full XY slab which contains 256^2 nodes. The decomposition is performed in units of nodes along the X axis. Each process is assigned an integer number of "lines" of 256 nodes. The first 2,536 processes are assigned 22 lines while the remaining processes are assigned 21 lines. Since this data structure is stored based on column-major ordering, this local node assignment to each process is contiguous.

Additionally, each process stores a set of ghost nodes that consist of nodes adjacent to their assigned nodes.

Each process's nodes (ghost and local) are stored in a contiguous array that is a subset of the global grid. This data structure is also ordered column-major ensureing that the local nodes are contiguous. This data structure is shown in Figure 3.

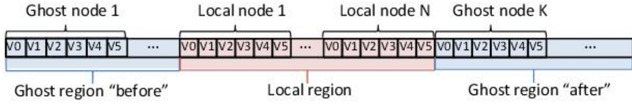

Figure 3: A representation of the local process's data structure. The local and ghost nodes are labeled.

The output file is in Tecplot binary format [13], which requires information from process 0 for section headers and information from all processes for grid and node data. Figure 4 shows the overall structure of the output file which includes data from all processes. The grid data, which consists of the node connectivity and X,Y and Z node coordinates, are output only for the first time-step (t_1). Every time step (t_x) includes a header and the values of all node variables. Each data section is ordered column-major, and each process supplies the values for its local nodes. MPI-IO [9, 10, 11] is used to write this file.

4.1 Optimization

The Tecplot binary output file can be divided into sections that correspond to each time step. The first time step is substantially different than subsequent time steps due to the addition of grid information. As a result, two subroutines are utilized to write this output file - one for the first time step and another for subsequent time steps. Originally, these subroutines opened the output file in append mode, wrote data, and closed the file. This mechanism, though effective, requires excessive overhead due to the many file opens, seeks, and closes. This overhead is reduced by keeping the output file open between time steps, which is implemented by moving the file open and close outside of these subroutines.

The header for the first time step consists of various data types of varying number, where N is the number of time steps: 8(MPI_CHAR), 52(MPI_INT), N[1(MPI_FLOAT), 13(MPI_INT), 1(MPI_DOUBLE), 11(MPI_INT)], 2(MPI_FLOAT), 12(MPI_INT), and 18(MPI_DOUBLE). The header for the subsequent time steps consists of 1(MPI_FLOAT), 21(MPI_INT), and 12(MPI_DOUBLE). These headers are only written by process 0 using a write call for each element (i.e. for N=1, 118 individual MPI_File_write calls are used for the first time step). This pattern generates a large number of very small writes. After optimization, a single MPI_File_write call is used for each data type and structure. Six write calls are used for the first time step and three calls are used for subsequent time steps. A struct and corresponding MPI derived data type are used for the N[1(MPI_FLOAT), 13(MPI_INT), 1(MPI_DOUBLE), 11(MPI_INT)] section of the header.

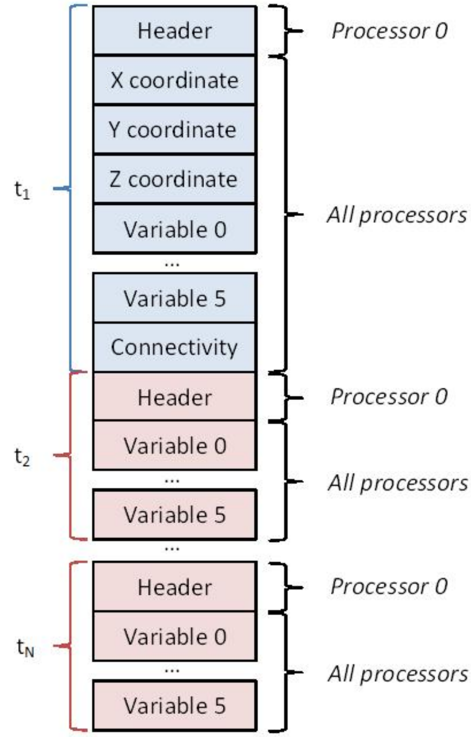

Figure 4: A representation of the Tecplot binary output file format.

The data sections (coordinates, connectivity, and variables) are written by all processes. These sections are ordered like the global data as a column-major array, in which each process's data is a contiguous subset. In the original routine, each process calls MPI_File_seek to move to the location in the global data set where its data begins. Then, since each process's data also contains ghost nodes and other variables which are not written, the array indices are looped over with a stride to select a particular variable in order to check whether each is a ghost node. If the index corresponds to a local node, it is written. A second seek is then necessary to align each process to the end of the data set. The result of this pattern is that a large number of small writes are performed, and seeks are necessary before and after each process's writes.

To remove the file seeks, a MPI derived data type is created via MPI_Type_create_subarray which defines the

location in the global data set for each process's data. This is implemented using MPI_File_set_view. Additionally, a second MPI derived data type is created via a combination of MPI_Type_vector to select a single variable from the array and MPI_Type_create_indexed_block to select the set of local nodes from the array. This allows each data set to be written with a single MPI_File_write call.

Utilizing a grid of 256^3 nodes and 3,000 processes, the amount of data written by each process, per data set, is much less than 1 MB. The applied optimizations allow the option to utilize collective writes in MPI-IO via MPI_File_write_all calls. This also enables the use of collective buffering within MPI-IO in order to aggregate I/O requests from multiple processes which increases the amount of data written in a single write.

Table 3: Comparison between Original and Optimized I/O in Application 3

		Bandwidth (GB/s)	
Time Step	Data (GB)	Original	Optimized
1	1.62	$3.47\text{x}10^{-02}$	8.15
2	0.75	$2.18\text{x}10^{-02}$	4.22
3	0.75	$1.91\text{x}10^{-02}$	5.39
4	0.75	$1.74\text{x}10^{-02}$	3.28
5	0.75	$2.18\text{x}10^{-02}$	4.54
6	0.75	$2.05\text{x}10^{-02}$	3.23
7	0.75	$2.01\text{x}10^{-02}$	4.85
8	0.75	$1.79\text{x}10^{-02}$	4.56
9	0.75	$2.59\text{x}10^{-02}$	4.79
10	0.75	$2.62\text{x}10^{-02}$	4.03

4.2 Results

The benchmark case utilizes 3,000 compute cores and performs 10 time steps. A single file is produced which contains all 10 time steps. This file is striped across 160 OSTs with a stripe size of 1 MB. The I/O time is determined by timing only the write calls and summing when appropriate. The maximum process time is taken as the I/O time. Table 3 shows the I/O bandwidth obtained per time step for both the original and optimized application code. The performance improves by about a factor of 200 between the original and optimized application code. The majority of this improvement, a factor of 100, is obtained by the use of collective buffering within MPI-IO. This optimization would not have been possible without first addressing the use of application data buffers, file access, and MPI derived data types.

5 Conclusion

Three scientific applications are optimized to improve their I/O performance. Although this task is completed while keeping the application's data layout in memory and on-disk constant, substantial performance improvements are realized. Improvements in performance ranged from a factor of 2 to 200 due to optimizations such as more efficient use of MPI derived data types, data buffering, and MPI-IO collective operations. The results include a savings of about 2 hours in I/O time for a data-intensive application, the potential to remove a data compression step which seems less efficient than writing the uncompressed data, and a factor of 100 increase in I/O performance that would not have been possible without initial optimizations.

6 About the Authors

Lonnie D. Crosby is a computational scientist at the University of Tennessee's National Institute for Computational Sciences (NICS) located at Oak Ridge National Laboratory (ORNL). Lonnie has a Ph.D. in Chemistry from The University of Memphis located in Memphis, TN. He may be contacted at Oak Ridge National Laboratory, P.O. Box 2008 MS6173, Oak Ridge, TN 37831-6173, Email: lcrosby1@utk.edu.

R. Glenn Brook is a computational scientist at the University of Tennessee's National Institute for Computational Sciences (NICS) located at Oak Ridge National Laboratory (ORNL). Glenn has a Ph.D. in Computational Engineering from the University of Tennessee at Chattanooga. He may be contacted at Oak Ridge National Laboratory, P.O. Box 2008 MS6173, Oak Ridge, TN 37831-6173, Email: glenn-brook@tennessee.edu.

Bhanu P. Rekepalli is a computational scientist at the University of Tennessee's National Institute for Computational Sciences (NICS) located at Oak Ridge National Laboratory (ORNL). Bhanu has a Ph.D. in Computer Engineering from The University of Tennessee, Knoxville. He may be contacted at Oak Ridge National Laboratory, P.O. Box 2008 MS6173, Oak Ridge, TN 37831-6173, Email: bhanu@utk.edu

Mikhail Sekachev is a graduate assistant at the University of Tennessee's National Institute for Computational Sciences (NICS). He is currently pursuing a doctorate degree in mechanical engineering at the University of Tennessee located in Knoxville, TN. He may be contacted by email at mikhail@utk.edu

Aaron D. Vose is a research associate at the University of Tennessee's Joint Institute for Computational Sciences (JICS) located at Oak Ridge National Laboratory (ORNL). Aaron has a B.S. in Computer Science from the University of Tennessee in Knoxville, TN. He may be contacted by e-mail at avose@eecs.utk.edu.

Kwai Wong is a computational scientist at the University of Tennessee's Joint Institute for Computational Sciences (JICS) located at Oak Ridge National Laboratory (ORNL). Kwai has a Ph.D. in Engineering Science from the University of Tennessee, Knoxville. He may be contacted at Oak Ridge National Laboratory, P.O. Box 2008 MS6173, Oak Ridge, TN 37831-6173, Email: kwong@utk.edu.

References

[1] Larkin, J; Fahey, M. "Guidlines for Efficient Parallel I/O on the Cray XT3/XT4." *Cray User Group Inc.: Conference Proceedings*, **2007**.

[2] Crosby, L. D. "Performance Characteristics of the Lustre File System on the Cray XT5 with Respect to Application I/O Patterns." *Cray User Group Inc.: Conference Proceedings*, **2009**.

[3] Kraken National Institute for Computational Sciences. http://www.nics.tennessee.edu/ computing-resources/kraken (accessed May 6, 2011).

[4] Sun Microsystems "Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System." *White Paper*, **October 2008**.

[5] Wong, K. L.; Baker, A. J. "A Modular Collaborative Parallel CFD Workbench." *J. Supercomputing* **2002**, *22*, 45 – 53.

[6] Cui, Y.; Olsen, K. B.; Jordan, T. H.; Lee, K.; Zhou, J.; Small, P.; Roten, D.; Ely, G.; Panda, D. K.; Chourasia, A.; Levesque, J.; Day, S. M.; Maechling, P. "Scalable Earthquake Simulation on Petascale Supercomputers." *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, **November 2010**.

[7] Altschul, S. F.; Gish, W.; Miller, W.; Myers, E. W.; Lipman, D. J. "Basic local alignment search tool." *J. Mol. Biol.* **1990**, *215* (3), 403 – 410.

[8] Rekepalli, B.; Vose A. "Petascale Genomic Sequence Search." Presented at Proceedings of The 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Newport Beach, CA, May 23 – 26, 2011.

[9] Thakur, R.; Gropp, W.; Lusk, E. "A Case for Using MPI's Derived Datatypes to Improve I/O Performance." *Proceedings of the IEEE/ACM Conference on SuperComputing: High Performance Networking and Computing*, **November 1998**.

[10] Thakur, R.; Gropp, W.; Lusk, E. "Data Sieving and Collective I/O in ROMIO." *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, **February 1999, 182 – 189**.

[11] Cray Inc. "Getting Started on MPI I/O." *Cray Doc S-2490-40*, **December 2009**.

[12] Deutsch. P. et al. "Zlib compressed data format specification version 3.3." *Informational Memo Aladdin Enterprises*, **1996**.

[13] Tecplot 360™ 2008 Data Format Guide, Appendix A: Binary Data File Format. ftp://ftp.tecplot.com/pub/doc/tecplot/360/dataformat.pdf (accessed May 6, 2011).