A Lightweight Scalable I/O Utility for Optimizing High-End Computing Applications

Shujia Zhou[◊*], Bruce H. Van Aartsen[*], and Thomas L. Clune
NASA Goddard Space Flight Center, Greenbelt, MD USA 20771
Shujia.Zhou@nasa.gov

Abstract

Filesystem I/O continues to be a major performance bottleneck for many High-End Computing (HEC) applications and in particular for Earth science models, which often generate a relatively large volume of data for a given amount of computational work. The severity of this I/O bottleneck rapidly increases with the number of processors utilized. Consequently, considerable computing resources are wasted, and the sustained performance of HEC applications such as climate and weather models is highly constrained. To alleviate much of this bottleneck, we have developed a lightweight software utility designed to improve performance of typical scientific applications by circumventing bandwidth limitations of typical HEC filesystems. The approach is to exploit the faster inter-processor bandwidth to move output data from compute nodes to designated I/O nodes as quickly as possible, thereby minimizing the I/O wait time. This utility has successfully demonstrated a significant performance improvement within a major NASA weather application.

1. Introduction

High-end computers typically have an I/O system as illustrated in Fig. 1 [1,2]. Between an application and I/O hardware, there are three software layers for the application to utilize I/O: (1) The high-level library maps application abstraction to a structured, portable file format. (2) The middleware layer deals with organizing access by many processes. (3) The parallel file system provides efficient access to data. Each layer offers potential for performance optimization.

In this paper we will focus on the layer of the high-level I/O library. One reason is that many climate and weather prediction systems have important common characteristics that are more easily and efficiently coped with by a high-level I/O library. First, they periodically write out a huge amount of data (10-25% of the application memory) in the form of multidimensional arrays that are each about 1% of the total memory. These applications are also highly synchronized – processors exchange information with their neighbors frequently during each time step. Thus, any delay on one processor, such as waiting for an I/O operation to complete, forces all other processors to wait as well. Operational weather models need to write simulation data to disk for analysis and forecasting within a short time window. Moreover, the desire for high-resolution simulations and the trend toward a rapidly-increasing number of available computational nodes, such as the 100,000 processors in IBM's BlueGene/L, further burdens the I/O system [3]. The comparatively slow performance of the I/O system forces compute nodes to remain idle for a significant time while writing the data to disk. For example, NASA's GEOS-5 atmospheric model takes ~25% of the total execution time, to output the data of a one-day simulation with a 720x360x72 grid using 120

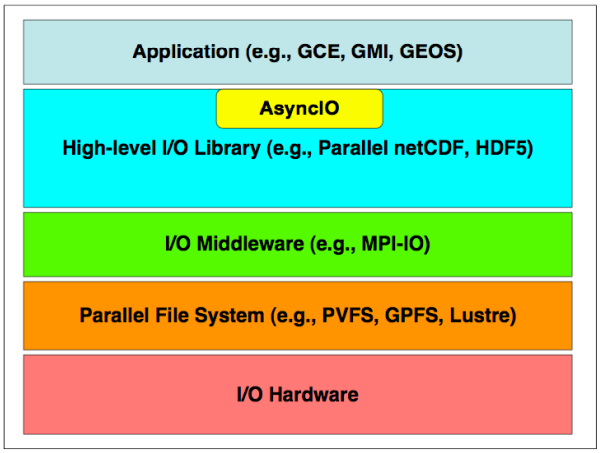

Figure 0. A representative I/O system architecture in a high-end computer

◊ Corresponding author

* Employee of Northrop Grumman Corporation

processors on an SGI Altix.

Because high-end platforms become obsolete very quickly (typical lifetimes are about 3 years), model developers have a strong incentive to prefer standard and stable programming interfaces to avoid recoding. Therefore, an enduring I/O software package must (1) have an abstract interface which isolates the users from specific vendor file systems, (2) be portable across major HPC architectures, and (3) require minimal modifications to the application for adoption.

Numerous approaches to alleviating I/O performance bottlenecks in HPC applications are in active use today. Here we consider some of the more common approaches and address the weaknesses for these approaches for our particular application space.

Existing low-level parallel filesystem software implementations, such as the Parallel Virtual Filesystem (PVFS) [4], benefit from additional memory (DRAM) on the I/O nodes to cache output data. This memory can be used to buffer write requests and thereby release the compute nodes from I/O operations more quickly than if only direct disk access was available. However, this feature is controlled by the operating system and not typically tunable for a specific user application. For a large cluster in use by a large number of varied applications, the system parameters are invariably suboptimal for some applications.

Some high-level parallel I/O filesystem software implementations exploit special hardware capabilities such as shared-memory. For example, a custom layer was developed at NASA Ames Research Center using the shared-memory capabilities of the SGI Altix architecture to pull state information directly from the compute nodes without interrupting computations. But this approach is obviously limited to similar shared-memory computer systems. Indeed, part of our motivation has been to enable similar capabilities on distributed memory architectures, albeit with relatively higher complexity.

Another I/O approach which is well suited to some architectures is to let each compute node write the data out to its local disk. The data on those disks can then be gathered with special software tools performed by another set of processors either as a background process or during postprocessing. For example, NASA Goddard's Land Information System (LIS) uses a customized script to access the distributed data [5] and a NOAA system uses netCDF [6] to store the decomposition information and collect the distributed data with the specially developed tool after the simulation is done. The advantage is that relatively minor modifications to the source code are involved. However, this implementation only benefits the applications that use netCDF. The efficiency can also be an issue. In the extreme case, on some computer systems, there is a locking mechanism for accessing data on the disks. For large numbers of processors (streams of data) this may result in very high levels of contention causing the I/O processes to slow to a crawl. More commonly though, we expect filesystems to be robust under this mode of operation, but provide rather limited scalability.

Improving I/O performance is also financially rewarding, due to the relative expense of system I/O nodes. For example, on the recently installed NASA Center for Computational Sciences (NCCS) Linux cluster, Discover, an I/O node is double the price of a compute node (and up to 5 times the price if disks are included).

2. Design

Our parallel asynchronous I/O tool, AsyncIO, addresses all the above issues for the bursty I/O operations of climate and weather forecasting applications. Recognizing that the inter-processor data transfer rate is much higher than the disk writing rate, AsyncIO exploits this rate disparity to move output from the compute nodes to designated I/O nodes at maximum speed, dramatically reducing the I/O wait time. The compute nodes are then free to resume calculations while data buffered on the I/O nodes continues writing to disk at the (slower) sustained speed allowed by the disk systems.

Please note that in the context of AsyncIO, the term "I/O nodes" actually refers to compute nodes being used for I/O. With system management support, actual system I/O nodes and other special nodes can also be used as AsyncIO I/O nodes.

This approach assumes that the memory required to store a single I/O "burst" is much smaller than the aggregate memory used by the entire model. Otherwise, the number of I/O nodes would be prohibitively large and the strategy would be self defeating. Some scientific and engineering applications require periodic dumps of essentially the full state of the application and are therefore poorly suited to our approach. For such cases there is no alternative to expensive, highly scalable parallel filesystems. However, our experience suggests that most Earth science applications possess characteristics favorable to our design.

To optimize the overall performance, it is necessary to enable an application to control when to flush the data from I/O nodes. In this way, an application can balance the data flow to fit the bandwidth of the inter-processor network, the number of I/O nodes, the memory size of the I/O nodes, and the disk speed. In

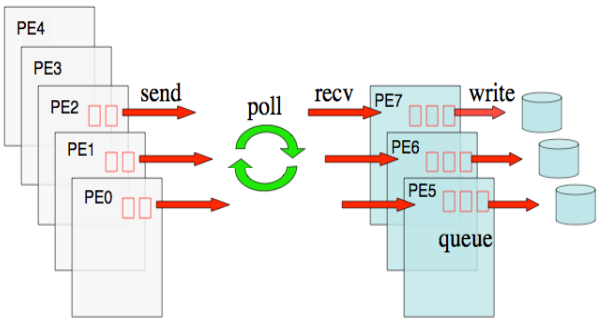

Figure 0. System architecture of application-controlled parallel asynchronous I/O

addition, this approach streamlines the I/O library and reduces its impact on application recoding. The I/O node memory can be increased by either employing a processor with more memory or by using multiple processors.

To accomplish these requirements, we decompose the process of writing data onto a disk, as illustrated in Fig. 2: (1) Send data from a compute node to a corresponding I/O node via the inter-processor network. This step takes advantage of the bandwidth of the inter-processor network, which greatly exceeds that of the filesystem. (2) Store the data in a queue. Using a queuing mechanism allows optimizing the amount of data cached in an I/O node, based on available memory at the node. (3) Pull the data out of the queue. (4) Write the data onto disk. Steps 1 and 2 are done in one command call while steps 3 and 4 are completed in another command call. This arrangement allows a user to make a decision when to write the data to disk. In addition, the size of data in the queue can be adjusted based on the available memory in the I/O node.

A polling mechanism is used between Send and Receive operations to support various data types in a flexible and extensible manner. Moreover, the code is written in standard MPI and Fortran 95 (F95) to ensure portability, and exploits advanced features of F95 to provide a user-friendly interface. Finally, sending data in parallel from multiple compute nodes to corresponding I/O nodes is implemented to aggregate the bandwidth of the inter-processor network. This feature allows the system core to increase its capability of buffering more data by increasing the number of I/O nodes. Our preliminary test results show that the parallel data transfer rate indeed increases in a scalable way.

To simplify AsyncIO use in a production-quality application, we have overloaded the primary interfaces to provide:

- Support for scalars as well as 1D, 2D and 3D arrays
- Support for integer, real, and double precision data types
- Support for simultaneous access to multiple files

Our design philosophy is lightweight and minimally intrusive. Not only does this approach reduce the software burden of adopting AsyncIO, but it also reduces the risk of subtle incompatibilities with other frameworks such as Earth System Modeling Framework (ESMF) [7,8,9]. An application based on ESMF such as the GEOS-5 suite of weather models can still use AsyncIO to optimize its I/O performance. A sketch of such an implementation is as follows. In the ESMF application driver, m processes are requested. Among m, p processes (i.e., *0, 1, ... p-1*) are assigned to an ESMF component called *A* for the original model simulation, while n processes (i.e., *p, p+1, ... m-1*) are assigned for the I/O operation used in AsyncIO. AsyncIO will create two communicators: one for the processes in the model and the other for the I/O processes. After that, the polling process is initiated, which waits for the data transferred from the ESMF component. When the application completes sending the entire burst of data, the user signals the I/O nodes to begin sending data to disk. (Eventually AsyncIO will automatically handle flushing data to disk.) As shown in Fig. 3, only a few straightforward changes are involved in using AsyncIO in an ESMF application.

3. Results

3.1. AsyncIO itself

We have performed a series of performance tests on AsyncIO for an output pattern typical of weather and climate applications. Namely, 10 single-precision arrays of 1440x720x70 (~290MB per array) are written to disk consecutively. A test run on the NASA NCCS SGI Altix, which has unidirectional bandwidth of 3.2GB/s and memory per processor of 2GB, shows that the speed of writing these arrays to a disk with a single processor is ~298MB/s. With AsyncIO, an effective bandwidth of 590MB/s was achieved by sending the data from a compute node (PE 0 on Fig. 2) to an I/O node (PE 5). We further optimized the AsyncIO queue by using F95 pointers, which eliminates unnecessary copies of the data, and were then able to achieve ~735MB/s on the Altix, which is a 2.5X improvement.

Since the above test exceeded the available DRAM in one node of NCCS's HP AlphaServer sc45 (2GB),

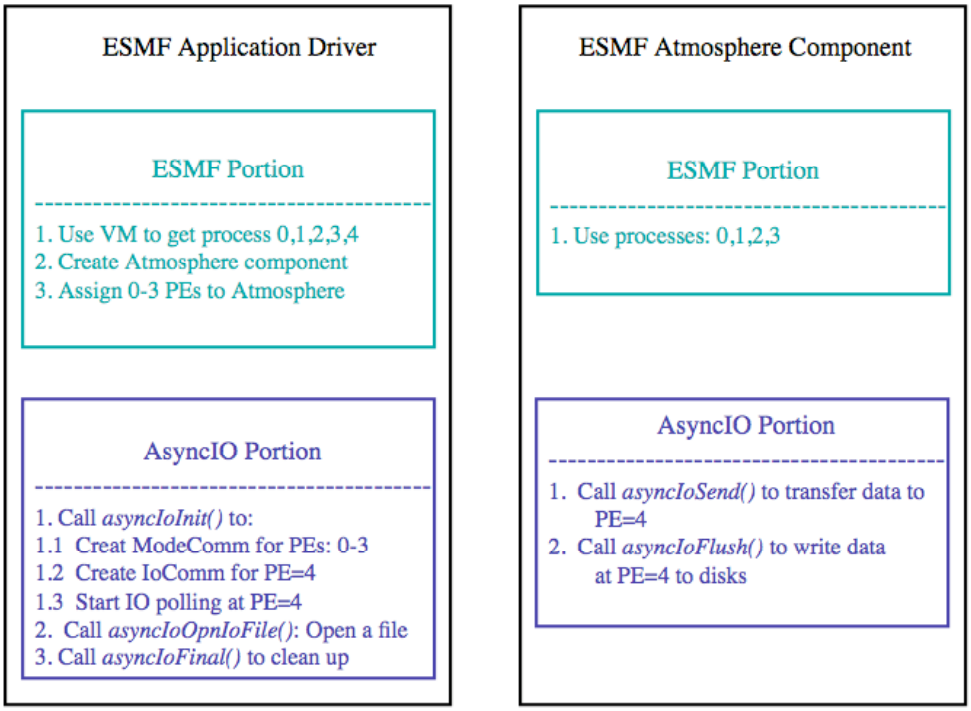

Figure 0. Using AsyncIO in an ESMF application

we performed a scaled-down test on that system, using 8 3D single-precision arrays of 1024x1024x41 (~1.37GB). With AsyncIO, our tests achieved ~121MB/s, which is ~3X improvement over writing data directly to a disk (~40MB/s). To aggregate the inter-processor bandwidth, AsyncIO supports the operation of sending the data in parallel from compute nodes to the corresponding I/O nodes. For example, sending 8 3D single-precision arrays of 1024 x 1024 x 41 achieves 1.7X and 2.3X improvement for two and three I/O processors, respectively. These preliminary results on AsyncIO itself clearly indicate that AsyncIO is capable of considerably increasing effective I/O performance.

We have also systematically evaluated the performance of AsyncIO on a more recent NCCS Linux cluster, Discover, which has 4GB/s memory bandwidth per core and 10Gb/s InfiniBand connections. Discover provides 4GB memory per node, though just under 3GB is user-accessible, so test arrays were kept below 2.4GB. These tests were run under "real world" conditions, competing for bandwidth with whatever processes were currently running on Discover. Under these conditions, AsyncIO delivered data transfer rates that averaged as high as 805MB/s for one set of tests, though the performance varied by up to 10%. The tests were run for various array sizes and processor configurations, and AsyncIO typically yielded effective data rates 4 to 5 times higher than the intrinsic file I/O, where the fastest write to disk was measured at 209MB/s. For general interest, further tests were run using a very small array (16x16x20). As expected, these tests showed no significant benefit with AsyncIO and speeds often varied wildly, apparently dependant on I/O cache utilization by other applications running on the platform. Fig. 4 below shows results on Discover for an array size typically used by the NASA Goddard Cumulus Ensemble (cloud model).

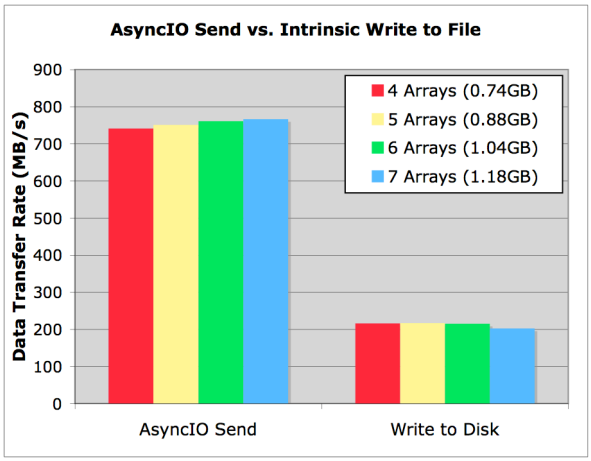

Figure 0. AsyncIO data transfer vs. standard write speeds on the Discover cluster

3.2. Integration with a model

To investigate how AsyncIO performs in a production-quality model, we chose to integrate it into the Goddard Cumulus Ensemble (GCE) model, which is used to simulate and study cloud formation processes as well as to improve the fidelity for global circulation models [10]. The integration of AsyncIO into GCE was very straightforward. The only notable obstacle was the deeply nested interface. We chose to use a global variable to share the MPI communicator rather than add an extra argument to a large number of interfaces.

To measure the I/O performance, we have run a series of simulations with three grid resolutions, 516x516x41, 1028x1028x41, and 1540x1540x41 (the memory-limited maximum resolution), on NASA's Columbia (SGI Altix) supercomputer. In the following, we discuss the performance measured for the case:

- Model time step is 6 seconds.
- Variables of microphysics, dynamics, and statistics are output every 20 time steps.
- 32 processors are used in the GCE standalone, while 37 processors are used in the GCE with AsyncIO, where 5 compute processors are used as I/O processors to store the 5 I/O files, respectively.

To integrate AsyncIO, we replaced each Fortran write operation with AsyncIO's send() command and its subsequent flush (to disk) command. During integration, an additional optimization operation was also carried out. Namely, write operations were grouped into two sets: microphysics and dynamics. Each group was collectively cached and then flushed to disk.

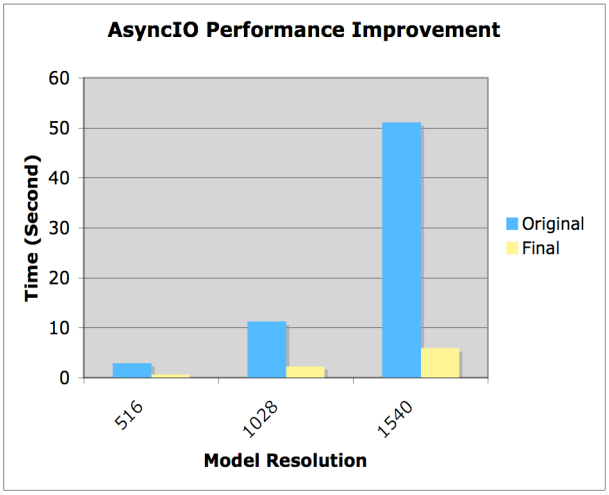

Figure 0. The I/O performance gain of using AsyncIO in the microphysics portion of GCE

Fig. 5 shows the performance gain of AsyncIO in the GCE Microphysics portion, where the "Original" time shows how long the compute node paused to perform a standard Fortran write. The "Final" time displays the dramatically shorter time required to execute the AsyncIO send() command, allowing compute nodes to resume work sooner. It is clear that the improvement increases greatly as the resolution increases from 516 to 1540, yielding an 8.6X improvement at the highest resolution.

4. Discussion

So far we have demonstrated that this lightweight utility can be very effective in improving the I/O performance. However, several features can be added to AsyncIO to improve efficiency and expand its applicability. For example, some simple enhancements might include (1) optimizing the pace of writing data from I/O processors to disk, or (2) making the gather operation output data directly to I/O processors rather than the root of compute processors. In the following, we discuss several potential enhancements in detail.

4.1. I/O data format

In the climate and weather forecasting community, there are three major data formats in use: raw binary (Fortran or C), netCDF [6], and HDF [11]. HDF and netCDF enhance portability of data sets by providing platform independent formats and also embed extremely valuable metadata (e.g., variable names, units, array dimensions, grid information, and time stamp). It should be noted that these capabilities do incur a nontrivial performance penalty on many platforms, but that the benefits for community science efforts generally outweigh such concerns. The AsyncIO application programming interface (API) could be augmented to facilitate using netCDF and HDF formats in a relatively straightforward manner. Additional interfaces would pass metadata to the I/O nodes, which would in turn access the filesystem exclusively through the netCDF or HDF libraries.

4.2. Parallel I/O

While both netCDF and HDF have variants which support parallel I/O [12], not to mention MPI-IO [13], we expect that there will remain scenarios where the fundamental bottleneck remains the finite amount of available disk cache. However, when using multiple I/O nodes within AsyncIO, these parallel I/O layers

could be leveraged to reduce the time to flush data to disk. This would only be of significant benefit in those cases where the frequency of I/O bursts from the application exceeds the sustained rate of serial I/O access to disk. Such scenarios might include the creation of high temporal resolution "movies".

A more flexible and general approach to allow a user to customize file data formats would be to provide an API for an application to pass the handle (procedure pointer) of its desired I/O subroutine to AsyncIO. AsyncIO can invoke the associated user function and allow it to manage the migration of the data to files. In this way, a user can control the final output data format while still obtaining the performance improvements from the asynchronous nature of the I/O utility. Because the application binary executable is generally identical on compute and I/O nodes, this approach would be quite portable, though technically not guaranteed to work on all platforms/compilers.

AsyncIO focuses on moving data out of compute nodes without immediately flushing it to disk, unlike the MPI IO method of collectively writing distributed data to disk. As such, we believe that AsyncIO can take advantage of MPI IO to further speed the process of writing data from AsyncIO's I/O nodes to disk.

4.3. Optimizing parallel data transfer between compute and I/O nodes

Transferring the data distributed at ***m*** nodes to the ***n*** nodes is a so-called ***m x n*** problem [14]. To ensure minimum impact on the applications, we can use the application parallel communication functions to gather data from ***m*** compute nodes to one compute node. In this way, the user can continue using the data orchestration already in place. When the total amount of data cannot be accommodated in one compute node, appropriate variables will be gathered to another compute node. This redirection will continue until all the output data are accommodated in the ***n*** compute nodes. Then the data in those ***n*** compute nodes will be transferred in parallel to their corresponding ***n*** I/O nodes as shown in Fig. 2 (where ***m***=5 and ***n***=3). We plan to look into the best scheme to optimize this parallel data-transferring operation. In addition, we hope to investigate tools such as the Model Coupling Toolkit [15, 16] and Intercomm [17] for supporting or enhancing this feature.

4.4. AsyncIO benefits on multi-core computers

Computer clusters with dual-core processors, such as NASA NCCS Discover, are currently serving applications at NASA. Quad-core chips are now being used in newer computer clusters and chips with more than 4 cores will soon follow.

The multicore computer architecture inevitably adds to the application developers' burden for achieving the highest I/O potential performance since the number of cores added into a chip would increase faster than the bandwidth improvement of the shared I/O port. AsyncIO allows a user to ensure the I/O processors are well-separated from each other so that local contention for bandwidth can be minimized. In addition, AsyncIO also allows us to choose the I/O processors that have higher sustained bandwidth to/from the disk subsystem. We will continue following the development of multi-core computer architectures and adapt the AsyncIO architecture accordingly.

4.5. Usability and release status

Since our design principle was to minimize the impact on the user's application and allow easy integration with MPI-based HEC applications, we were gratified to see that beta users adopted our API and integration procedure quickly. The use of standard F95 and MPI make AsyncIO fairly easy to add to an application's build process. We believe that any Fortran/MPI application that periodically outputs very large 1D, 2D, 3D arrays of integer, single, or double precision data types in unformatted sequential format should be able to employ AsyncIO to significantly increase its I/O performance.

We are currently pursuing an open-source release of AsyncIO. However, during the interim we are very interested in collaborating with potential developers and users to improve the I/O performance of their applications, and potentially enhance the features of AsyncIO.

5. Conclusion

Our application-controlled parallel asynchronous I/O library has demonstrated a significant increase in I/O performance. In addition, our minimal-intrusion user-interface design makes it easily integrated into production-quality code. Moreover, its implementation using standard features of Fortran and MPI make this library portable across various computer platforms.

6. Acknowledgements

We would like to thank A. Oloso and M. Damon for evaluating the PVFS filesystem and testing some of the AsyncIO code. We would also like to thank D. Duffy and J. Dorband for helpful discussions on the I/O system of NCCS's Discover cluster and the system

caching mechanism of PVFS, respectively. Finally we'd like to thank Wei-Kuo Tao for providing the NASA GCE code and Xiping Zeng for assistance in using the code.

7. References

[1] J. M. May, *Parallel I/O for High Performance Computing,* Morgan Kaufmann Publishers 2001.

[2] R. Latham et al., "Parallel I/O in Practice," tutorial in Supercomputing Conference, Tampa, November 13, 2006.

[3] R.A. Oldfield, P. Widener, A.B. Maccabe, L. Ward , T. Kordenbrock, "Efficient Data-Movement for Lightweight I/O, " 2006 IEEE International Conference on Cluster Computing, 25-28 Sept. 2006, pp. 1-11.

[4] PVFS, http://www.parl.clemson.edu/pvfs/

[5] Y. Tian, C. D. Peters-Lidard, S. V. Kumar1, J. Geiger, P. R. Houser, J. L. Eastman, P. Dirmeyer, B. Doty, J. Adams, *Computers & Geosciences*, in press.

[6] netCDF, http://www.unidata.ucar.edu/packages/netcdf/

[7] ESMF, http://www.esmf.ucar.edu

[8] C. Hill, C. DeLuca, V. Balaji, M. Suarez, A. da Silva, and the ESMF Joint Specification Team, "The Architecture of the Earth System Modeling Framework," *Computing in Science and Engineering*, Vol. 6, No. 1, 2004.

[9] S. Zhou et al., "Cross-Organization Interoperability Experiments of Weather and Climate Models with the Earth System Modeling Framework," *Concurrency and Computation: Practice and Experience*, 19(5):583-592, April 2007.

[10] The Goddard Cumulus Ensemble (GCE) model, http://atmospheres.gsfc.nasa.gov/cloud_modeling/models_gce.html

[11] HDF, http://www.hdfgroup.org

[12] Parallel netCDF, http://www.mcs.anl.gov/parallel-netcdf/

[13] W. Gropp, E. Lusk, and R. Thakur, "Using MPI-2: Advanced Features of the Message Passing Interface," MIT Press, Nov. 26, 1999.

[14] L.C. McInnes, B.A. Allan, R. Armstrong, S.J. Benson, D.E. Bernholdt, T.L. Dahlgren, L.F. Diachin, M. Krishnan, J.A. Kohl, J.W. Larson, S. Lefantzi, J. Nieplocha, B. Norris, S.G. Parker, J. Ray, and S. Zhou, "Parallel PDE-Based Simulations Using the Common Component Architecture," an invited chapter in the book "*Numerical Solution of Partial Differential Equations on Parallel Computers*," A. M. Bruaset, P. Bjorstad, and A. Tveito, editors, published by Springer-Verlag 2006.

[15] J. Larson, R. Jacob, E. Ong, "The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models." *2005: Int. J. High Perf. Comp. App.,* 19(3), 277-292.

[16] R. Jacob, J. Larson, E. Ong "MxN Communication and Parallel Interpolation in CCSM3 Using the Model Coupling Toolkit." *2005: Int. J. High Perf. Comp. App.,* 19(3), 293-307.

[17] Intercomm, http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/ic