

The Parallel-netCDF I/O Library

Robert Latham
Mathematics and Computer Science Division
Argonne National Laboratory

December 8, 2010

1 Synonyms

pnetcdf, High-level I/O Library

2 Definition

The serial netCDF library has long provided scientists with a portable, self-describing file format and a straightforward programming interface based on multidimensional arrays of typed variables (more detail in Section 3.2). Parallel-NetCDF provides an API for parallel access to traditionally formatted netCDF files. Parallel-NetCDF both produces and consumes files compatible with serial netCDF, while providing a programming interface similar, though not identical, to netCDF. Parallel-netCDF API modifications provide a more appropriate interface when expressing parallel I/O.

3 Discussion

Parallel-NetCDF is one of a family of libraries built to meet the needs of computational scientists, as opposed to lower-level I/O libraries that deal more with the details of underlying storage. Before going into more detail about Parallel-NetCDF, let us see how it fits in the bigger picture of scientific computing and application I/O.

3.1 Parallel-NetCDF and the I/O Software Stack

Today's leadership computing platforms contain hundreds of thousands of processors. In order to efficiently utilize these computers, applications use simplifying abstractions provided by tools and libraries. MPI libraries provide a standard programming interface for parallel computation on a wide array of hardware. Math libraries such as BLAS hide the details of CPU-specific optimizations. These lower-level math and communication libraries contribute to

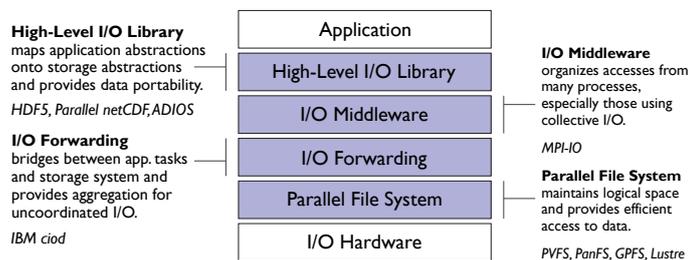


Figure 1: The I/O software stack. Parallel-NetCDF sits at the top, allowing applications to express data manipulations more abstractly while lower levels deal with coordination among multiple processes and maximize performance of storage devices.

an overall software stack built to allow developers to focus on the science behind their applications.

I/O performance on high-end computing platforms follows a similar story. Performance comes from aggregating many storage devices. In fact, CPU performance has consistently improved at a rate much faster than that of storage devices. This discrepancy makes the need for parallel I/O across multiple devices even more acute. Yet as the number of devices involved in I/O grows, coordinating I/O across those devices becomes more of a challenge. In the same way that communication and math libraries assist the parallel computation side of scientific simulations, an “I/O software stack” assists data management.

The large number of storage devices mentioned above forms the foundation of the software stack. Several additional layers provide abstractions to help make the task of extracting maximum performance accessible to applications programmers. The programming API and data model that Parallel-NetCDF presents to the scientific programmer are conceptually closer to the tasks common to scientific applications and hide many lower-level details. A discussion of the lower levels of the software stack, though hidden by Parallel-NetCDF, will provide a better appreciation of Parallel-NetCDF’s role at the upper level.

- Storage systems contain thousands of individual devices to increase the potential bandwidth. The *parallel file system* manages those separate devices and collects them into a single logical unit. The file system does lack some important features: typically file systems have no mechanism for coordinated I/O, and the data model is still fairly low level for computational scientists to use directly.
- The *I/O forwarding* layer typically exists only on the largest computer systems. Applications do not interact with this layer directly. Rather, this layer simplifies the scalability challenge: a large number of compute nodes communicate with a smaller number of I/O forwarding nodes, and these nodes in turn talk to the file system.

- The *middleware*, or *MPI-IO* [6], layer introduces more sophisticated algorithms tailored to parallel computing. This layer coordinates I/O among a group of processes, introducing collective I/O to the stack. MPI datatypes allow applications to describe arbitrary I/O access patterns. Applications can use the MPI-IO layer directly, but the programming model is still rather low-level and not a perfect fit to the needs of common applications. MPI-IO libraries, however, provide an excellent foundation for higher-level libraries such as Parallel-NetCDF.

Parallel-NetCDF sits atop these lower-level abstractions; it belongs to the *high-level I/O library* layer. Libraries in this layer introduce concepts such as multidimensional arrays of typed data, which are better suited to scientific applications. For example, a climate code can represent temperature in the atmosphere with a three-dimensional array – latitude, longitude and altitude – containing the number of degrees Celsius. In addition to array storage, applications can also provide descriptive information in the form of attributes on variables, dimensions, or the dataset itself. These libraries also define the layout of data on disk. These annotated, self-describing file formats make exchanging data with colleagues now and in the future much easier.

3.2 History

The netCDF library has long provided a straightforward programming API and file format for serial applications. In the summer of 2002, Argonne National Laboratory and Northwestern University started a joint effort to build a parallel I/O library while keeping the best parts of netCDF.

Parallel programs had been using netCDF for some time, though the serial library forced programmers to use serial approaches when writing out netCDF datasets. In one approach, each process writes out its own data set. This “N-to-N” model can put significant strains on both the file system and any postprocessing tools, especially since large systems today can have hundreds of thousands of processes. Another approach sends all data to a master I/O process, and that process in turn writes out the dataset. This “send-to-master” technique imposes an obvious bottleneck because all data must be funneled through a single process.

Without changing the existing netCDF file format, Parallel-NetCDF introduced a new programming API. This parallel-oriented API is not identical to serial netCDF, but is comfortable to anyone familiar with the existing serial netCDF API. The Parallel-NetCDF API allows computational scientists to write out datasets in parallel: all processes can participate in an “N-to-1” operation, producing a single dataset while leveraging a host of parallel I/O optimizations. Section 3.3 contains examples and more discussion of the Parallel-NetCDF API.

Parallel-NetCDF has become an important tool for high-performance I/O in the climate and weather domains. These domains have long-established netCDF-based workflows, using netCDF datasets for archiving, analysis, and data exchange. Since Parallel-NetCDF retains the same file format as netCDF,

researchers in climate and weather can make changes to their simulation codes while leaving other components of their workflow the same. Parallel-NetCDF is useful in other domains as well: the programming API and file format make it possible to deliver good parallel I/O performance without having to master all of MPI-IO.

3.3 The Parallel-NetCDF API and File Format

Discussions about Parallel-NetCDF need to cover two main points: the programming model and the on-disk data format. The API simplifies many MPI-IO concepts while retaining several key optimizations. The on-disk format affects I/O performance, but it also matters when exchanging data with collaborations and colleagues.

If applications were to use MPI-IO directly, the “basic type” they would build on would be a linear stream of bytes. In contrast, Parallel-NetCDF offers a more application-friendly model based on multidimensional arrays of typed data. Operations on these arrays could be as simple as reading the entire array, or more complicated such as having each process write a sub-cube representing a chunk of the Earth’s atmosphere. In addition to operations for manipulating data, the API contains routines to annotate the data in the file or the file itself. Such annotations may include timestamps, machine information, experiment or workflow information, or other provenance information to better understand how an application produced the data in this file. Section 3.3.1 goes into more detail about the programming interface as well as how Parallel-NetCDF interacts with the underlying MPI-IO library.

In addition to the programming interface, the file format plays a major role in how useful Parallel-NetCDF can be to application groups. Parallel-NetCDF applications create a self-describing, portable file with support for embedded metadata. API routines allow programs to query the contents of data files without any prior knowledge of the contents. The structure of the datasets extends to the representation of bytes on disk: even if a dataset is moved to a different machine, the library will still be able to understand that file. The IBM Blue Gene/P system, for example, has a 32-bit, big-endian architecture, while the Cray XT5 has a 64-bit little-endian architecture. Despite these differences, files created on one can be read on the other and vice versa.

Figure 2 depicts a simple Parallel-NetCDF application. A climate code wants to create a checkpoint file to store some key pieces of information. Atmospheric temperature is stored as a three-dimensional array of double-precision floating point values. Barometric pressure at the Earth’s surface requires only two dimensions and only single-precision floating points. Further, the application wants to ensure that future consumers of this dataset know which units these variables use. It therefore stores this information as an attribute on each variable. On disk, the dataset contains a small header describing the size and type of both variables and any attributes, followed by the actual data for the variables. This fairly simple file format comes with some restrictions, but the trade-off results in efficient file accesses for parallel I/O.

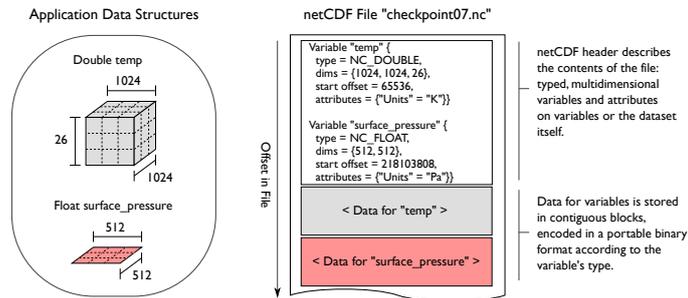


Figure 2: On the left, how an application views netCDF objects. On the right, how netCDF and Parallel-NetCDF store those objects on disk.

Parallel-NetCDF uses the same file format as netCDF, making it easier for applications groups to adopt Parallel-NetCDF. A programmer can change the computationally intensive simulation code to use Parallel-NetCDF while the other components of the workflow (e.g., visualization, analysis, archiving) can continue to use serial netCDF.

Annotated Examples Parallel-NetCDF shares many netCDF concepts but uses a slightly different API. The following simple example programs should make the differences clear.

3.3.1 Standard Interface

Figure 3 contains a full, correct Parallel-NetCDF program to create a simple dataset and write data into that dataset in parallel. This program, though brief, demonstrates many key features of the library.

Because Parallel-NetCDF relies on MPI-IO, the program must include the `mpi.h` header file (line 1) and initialize (line 9) and finalize (line 28) the MPI library.

The `ncmpi_create` routine (line 10) adds two arguments to netCDF's creation function: an MPI communicator and an MPI Info object. These two arguments do expose a bit of the underlying MPI library, but they give the programmer some flexibility and the ability to tune operations if needed. In this example, the communicator is just `MPI_COMM_WORLD`, or all processes in this MPI program. In some situations, though, the application may wish to have only a subset of processes participate in a dataset operation. The Info object in this example is empty, but Section 3.4 will cover some situations where the object is used.

Parallel-NetCDF, like netCDF, has a *bimodal* interface: when creating a dataset, an application starts in *define mode*, where it must first describe what variables it will write, the size and type of those variables, and any attributes.

```

1  #include <mpi.h>
2  #include <pnetcdf.h>
3
4  int main(int argc, char **argv) {
5      int ncfile, nprocs, rank, dimid, varid1, varid2, ndims=1;
6      MPI_Offset start, count=1;
7      char buf[13] = "Hello World\n";
8
9      MPI_Init(&argc, &argv);
10     ncmpi_create(MPI_COMM_WORLD, "demo.nc",
11                 NC_WRITE|NC_64BIT_OFFSET, MPI_INFO_NULL, &ncfile);
12
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
15
16     ncmpi_def_dim(ncfile, "d1", nprocs, &dimid);
17     ncmpi_def_var(ncfile, "v1", NC_INT, ndims, &dimid, &varid1);
18     ncmpi_def_var(ncfile, "v2", NC_INT, ndims, &dimid, &varid2);
19     ncmpi_put_att_text(ncfile, NC_GLOBAL, "string", 13, buf);
20
21     ncmpi_enddef(ncfile);
22
23     start = rank;
24     ncmpi_put_vara_int_all(ncfile, varid1, &start, &count, &rank);
25     ncmpi_put_vara_int_all(ncfile, varid2, &start, &count, &rank);
26
27     ncmpi_close(ncfile);
28     MPI_Finalize();
29     return 0;
30 }

```

Figure 3: Basic Parallel-NetCDF program using the standard interface. Each process collectively writes its MPI rank into two variables. For brevity, error checking/handling has been omitted.

This example defines a single dimension (line 16) and assigns that dimension to two variables (lines 17 and 18).

Datasets can contain a great deal of metadata. Both variables and dimensions have human-readable labels. In addition to those labels, attributes may be defined and placed on dimensions, variables, or the dataset itself. Line 19 places an attribute on the entire dataset.

In the example, line 21 calls `ncmpi_enddef` to switch from define mode to *data mode*. By asking the programmer to predefine variables and attributes, the library can allocate space with little overhead. Re-entering define mode can potentially trigger an expensive rewriting of the dataset. For a large class of problems, however, one knows ahead of time which information will be written out.

Lines 24-25 do the actual work of storing information into the dataset. Parallel-NetCDF provides one set of functions for uncoordinated *independent I/O* and another parallel set for coordinated *collective I/O*. This example uses the collective version (as shown by the `_all` suffix). Every MPI process participates in these writes, each writing out its MPI rank. In a traditional serial netCDF program, each rank would send data to a master process, and this process would in turn write out the information. This “send-to-master” model quickly becomes untenable, however, as the number of MPI processes increases and as the amount of memory available to each MPI process gets smaller.

The collective Parallel-NetCDF routines in turn call collective MPI-IO routines. MPI-IO collective routines can use powerful optimizations such as two-phase I/O [4] or data shipping [3]. It’s possible to use independent I/O with Parallel-NetCDF, but the opportunities for optimization are greatly limited in that case.

Once executed, this example will produce a file “demo.nc.” This dataset will contain two variables, as can be verified with either netCDF or Parallel-NetCDF utilities.

The Parallel-NetCDF standard interface shares much in common with the serial netCDF interface. The function name explicitly states whether the operation is a read or write (`_put` or `_get`), the type of access (e.g., `_var`, `_vara`), and the type (e.g., `_int`, `_float`, `_double`). For example, from the name alone a programmer knows that the function `ncmpi_put_vara_int` writes integers into a subarray.

A more realistic example This simple example, operating on a one-dimensional array, may not give the most insight into all the work going on behind the scenes. Figure 4 depicts a two-dimensional array of which this example program wants to only write a smaller subarray. Perhaps this array represents one frame of a movie and a parallel program renders portions of each frame.

Figure 5 contains the Parallel-NetCDF code needed to write the indicated region. First, the program must describe the entire variable: lines 6, 7, and 9 describe the overall shape of the variable and the type of data it will contain. Lines 13 and 14 set up the shape of the desired selection or subregion of the

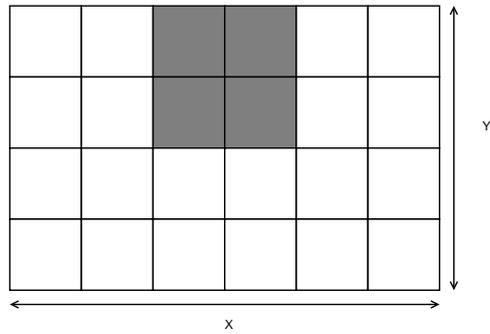


Figure 4: Writing a subarray of a two-dimensional array

```

1  ...
2  #define NDIMS 2
3  int dims[NDIMS], varid1, ndims=NDIMS;
4  MPI_Offset start[NDIMS], count[NDIMS];
5
6  ncmpi_def_dim(ncfile, "y", 4, &(dims[0]));
7  ncmpi_def_dim(ncfile, "x", 6, &(dims[1]));
8
9  ncmpi_def_var(ncfile, "frame", NC_DOUBLE, ndims, dims, &varid1);
10
11 ncmpi_enddef(ncfile);
12
13 start[0] = 0; start[1] = 4;
14 count[0] = 2; count[1] = 2;
15 ncmpi_put_vara_double_all(ncfile, varid1, start, count, data);
16 ...

```

Figure 5: Writing the selected portion of the above array.

```

1  /* function prototype */
2  int ncmpi_put_vara_all(int ncid, int varid,
3     /* start and count describe access in file */
4     const MPI_Offset start[], const MPI_Offset count[],
5     /* 'buf' 'bufcount' and 'datatype'
6     describe data in memory */
7     const void *buf, MPI_Offset bufcount, MPI_Datatype datatype);
8
9  ...
10 start = rank;
11 ncmpi_put_vara_all(ncfile, varid1, &start, &count,
12     &rank, count, MPI_INT);
13 ...

```

Figure 6: Prototype for and usage of one of the Flexible Data Mode routines.

variable and pass these arguments as parameters to the call at line 15.

These few lines of Parallel-NetCDF code trigger a lot of activity behind the scenes. Even though the desired region is logically contiguous, when the array is stored on disk, this selection will end up scattered across the file in a noncontiguous access pattern. Fortunately for Parallel-NetCDF, MPI-IO makes it easy to describe these accesses with a file view.

Storage systems yield the best performance with a few large, contiguous accesses. The MPI-IO library will apply several optimizations to transform a noncontiguous access into one more likely to give good bandwidth. The Parallel-NetCDF call in this example is collective, which means that the lower MPI-IO layer can in turn use collective I/O optimizations: the MPI-IO library can communicate with the other processes participating in this I/O call and rearrange the requests. To the file system, this subarray access will end up looking like a more performance-friendly contiguous request.

While this discussion glosses over many of the details, the important point is that a few Parallel-NetCDF functions convey a great deal of information to the lower levels of the I/O stack. Applications using Parallel-NetCDF thus get the benefits of a self-describing portable file format and a relatively straightforward API while still achieving performance without requiring the application programmers to master MPI-IO.

3.3.2 The Flexible Interface

Parallel-NetCDF provides an additional set of routines to allow for even more freedom in describing an application’s data model. In Figure 6, a write operation from Figure 3 is converted to the Flexible Data Mode interface. This interface allows an application to use MPI datatypes to describe how information is laid out in memory.

The datatype in this example is simple (`MPI_INT`), but a more complicated MPI datatype might, for example, write out all the data in a multidimensional array while skipping over the “ghost cells.” Ghost cells, copies of data belonging to other MPI processes, are used to optimize communication in nearest-neighbor simulations and do not actually need to be written to disk. In many cases, the

```

1 NCMPI_Request requests[2];
2 int status[2];
3 ...
4 ncmpi_iput_vara_int_all(ncfile, varid1, &start, &count,
5                          &rank, count, &(requests[0]));
6 ncmpi_iput_vara_int_all(ncfile, varid2, &start, &count,
7                          &rank, count, &(requests[1]));
8 ...
9 ncmpi_waitall(2, requests);

```

Figure 7: The “implicit” nonblocking interface: no progress occurs in the background, but operations can be batched together when completed.

Flexible Data Mode interface permits an application to avoid an additional buffer copy before making a Parallel-NetCDF call.

3.3.3 Nonblocking Interface

Parallel-NetCDF further extends the traditional netCDF API through the introduction of nonblocking I/O routines. Programmers familiar with MPI nonblocking communication routines will note strong similarities between MPI nonblocking routines and those in Parallel-NetCDF. A program posts one or more operations and then waits for completion of those operations.

In Figure 7 the example from Figure 3 has been modified yet again, this time to use Parallel-NetCDF nonblocking operations. Throughout the I/O software stack, more information about I/O activity results in more opportunities for optimization. In this example, even though the calls operate on separate variables, those variables are still in the same dataset and appear to the MPI-IO layer as parts of a single file. Parallel-NetCDF can take these nonblocking requests and optimize them into a single, larger I/O operation [1].

Parallel-NetCDF can optimize these nonblocking requests in this way because the library makes no guarantees about when work will occur (sometimes called “make progress” in the MPI context). It may appear to the program that I/O work happens in the background when calling these nonblocking routines. Actually, when the operation is posted (lines 4 and 6), no work happens. Instead, the library defers all work until the application makes an additional “wait for completion” function call (line 9). Once Parallel-NetCDF has a list of all the outstanding operations, it can then construct a single I/O request encompassing all operations. Typically, storage systems perform better with larger I/O requests, so coalescing operations in this way can yield good performance improvements.

3.4 Tuning Parallel-NetCDF

Extracting good performance out of Parallel-NetCDF comes down largely on following a handful of best practices: perform collective I/O to a small number of files. Doing so provides the most information to the lower I/O software

stack layers, and allows for those layers to make as many optimizations as possible. For even more fine-tuning, the library does provide some additional tuning mechanisms for applications. For the most part, these tuning knobs help tweak parameters for layers farther down the software stack. The MPI Info parameter passed to the create and open calls can direct the optimizations the underlying MPI-IO library chooses to make. For example, on Lustre file systems file locks are exceedingly expensive. By setting the MPI Info hint `romio_ds_write` to `"disable,"` the ROMIO implementation of the MPI-IO library, the most common implementation on systems with Lustre installed, will avoid using locks. Knowing more about the characteristics of the storage system and the application workload will make the selection of appropriate hints easier and more productive. The available MPI-IO tuning parameters vary based on the MPI-IO implementation and file system used. Users should consult their MPI-IO library's documentation.

Often, the most effective way for a computational scientist to improve the I/O performance of a program is to enlist the aid of the Parallel-NetCDF community (see Section 5). If the scientist can remove the simulation and science aspects of the program, leaving only the representative data structures, the resulting "I/O kernel" becomes a valuable resource for exploring all tuning options available. I/O kernels leave no doubt as to what the scientist requires from the Parallel-NetCDF library and storage system. Because these small programs have few if any dependencies on additional libraries, they can become part of the library's correctness and performance tests, ensuring that modifications and improvements made to Parallel-NetCDF also benefit applications.

A discussion about tuning Parallel-NetCDF would not be complete without a few words concerning record variables. In Parallel-NetCDF and netCDF, variables can have an unlimited dimension. Often, variables that change over time use this feature: each iteration of the simulation can append data along the unlimited "time" dimension. When more than one variable contains an unlimited dimension, those variables are stored on disk in an interleaved fashion. The writing and reading of data interleaved in this way will often yield poor performance. Programmers must weigh the flexibility of record variable storage against the cost of performance. With an I/O kernel, Parallel-NetCDF experts can likely find a set of tuning parameters to mitigate some of the performance loss, and the project will be looking at more sophisticated ways of dealing with record variables in the future.

3.5 Conclusion

The Parallel-NetCDF library provides a more natural programming interface for high-performance storage systems. By implementing parallel I/O concepts in terms of multidimensional arrays, many computational science simulations are able to naturally express their data structures.

Parallel-NetCDF provides a further benefit in that it encapsulates a great deal of I/O expertise. Computer scientists working on high-performance storage may not know a great deal about weather, climate, combustion, or other

scientific domains but know much about how to get the best performance out of storage devices, the file system, and MPI-IO. Application scientists in turn specialize in modeling phenomena, developing numerical methods, and exploring other topics more relevant to computational science. Parallel-NetCDF and other high-level I/O libraries provide a middle ground where scientists and storage experts can come together to maximize productivity.

3.6 Related Work

The HDF5 library [5] was the first high-level I/O library to be built on top of MPI-IO. HDF5 provides a large and flexible API. For example, HDF5 applications do not need to enter a define mode to describe the variables and dimensions. HDF5 writes all metadata for the file on an as-needed basis. The trade-off for this flexibility is some additional overhead if the application is creating many new variables or other metadata-intensive workloads. HDF5 has the further benefit of allowing variables to grow without bound in any dimension. The HDF5 library allocates space for variables on demand. Consider the storage of a sparse matrix: only those values actually present in the matrix would be stored on disk. When multiple processes are updating variables in this way, however, this on-demand space allocation can result in some performance and consistency challenges. HDF5 currently forces all parallel metadata updates through rank 0, though the group is working on more sophisticated techniques.

More recently, the serial netCDF developers released netCDF-4 [7], a library with the netCDF programming API on top of the HDF5 file format. This project brought several HDF5 features to netCDF, including parallel I/O support, support for variables with multiple unlimited dimensions, and support for much larger variables. While introducing a new file format, it still can operate on older netCDF datasets. The netCDF-4 API does not have some of the more sophisticated Parallel-NetCDF features like the Flexible Mode interface or the Nonblocking interface.

4 Related Encyclopedia Entries

- Parallel-NetCDF fits squarely under the topic of *Parallel I/O*, designed with the goal of making parallel I/O accessible to a wider audience.
- It is closely related to *HDF5*, the first high-level I/O library. The two libraries make different design choices with regards to API and file format design. The friendly competition between HDF5 and Parallel-NetCDF ensures that both remain high-quality I/O libraries.
- Parallel-NetCDF sees most use in *Distributed Memory* environments, particularly those using message passing (though nothing precludes shared-memory platforms from using Parallel-NetCDF effectively).
- *MPI*, particularly the MPI-IO portion of MPI, provides the foundation on which Parallel-NetCDF is built.

- *Benchmarking* ties into Parallel-NetCDF in that correctly benchmarking Parallel-NetCDF requires understanding the entire I/O software stack. Parallel-NetCDF should yield performance close to that of straight MPI-I/O: should a benchmark show significantly higher or lower performance, that would certainly warrant further examination.
- The *Weather and Climate Simulation* fields make use of Parallel-NetCDF, particularly as per-node memory sizes become smaller and smaller. With less memory per-node, parallel approaches to I/O become important not just to enhance performance but to be able to run on these systems at all.

5 Bibliographic Notes and Further Reading

For more technical insight into Parallel-NetCDF, Jianwei Li et al.'s SC 2003 paper [2] presents the design choices made in developing the library; experimental results are also presented. The multivariable I/O optimizations are covered further in [1].

The Parallel-NetCDF website is www.mcs.anl.gov/parallel-netcdf. The site hosts pointers to additional Parallel-NetCDF papers and projects as well as links to production releases and the latest code.

The Parallel-NetCDF community of users and developers communicates on the parallel-netcdf@mcs.anl.gov mailing list. The quality of discourse on the list has been high from the beginning. Any topic related to Parallel-NetCDF is open for discussion.

Parallel-NetCDF developers regularly give tutorials and workshops covering the library.

Acknowledgments

This work was supported by the U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

<p>The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.</p>

References

- [1] Kui Gao, Wei keng Liao, Alok Choudhary, Robert Ross, and Robert Latham. Combining i/o operations for multiple array variables in parallel netcdf. In *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage, held in conjunction with the IEEE Cluster Conference, New Orleans, Louisiana, September 2009*.
- [2] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003: High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.
- [3] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Proceedings of SC2001*, November 2001.
- [4] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.
- [5] The HDF Group. HDF5. <http://www.hdfgroup.org>, 2008.
- [6] The MPI Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997.
- [7] Unidata. netCDF4. <http://www.unidata.ucar.edu/software/netcdf/index.html>.