Geosci. Model Dev., 7, 93–103, 2014
www.geosci-model-dev.net/7/93/2014/
doi:10.5194/gmd-7-93-2014

Geoscientific
Model Development
Open Access

A fast input/output library for high-resolution climate models

X. M. Huang, W. C. Wang, H. H. Fu, G. W. Yang, B. Wang, and C. Zhang

Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science, Tsinghua University, 100084, and Joint Center for Global Change Studies, Beijing, 100875, China

Correspondence to: X. M. Huang (hxm@tsinghua.edu.cn)

Received: 21 August 2013 – Published in Geosci. Model Dev. Discuss.: 13 September 2013
Revised: 30 November 2013 – Accepted: 3 December 2013 – Published: 14 January 2014

Abstract. We describe the design and implementation of climate fast input/output (CFIO), a fast input/output (I/O) library for high-resolution climate models. CFIO provides a simple method for modelers to overlap the I/O phase with the computing phase automatically, so as to shorten the running time of numerical simulations. To minimize the code modifications required for porting, CFIO provides similar interfaces and features to parallel Network Common Data Form (PnetCDF), which is one of the most widely used I/O libraries in climate models. We deployed CFIO in three high-resolution climate models, including two ocean models (POP and LICOM) and one sea ice model (CICE). The experimental results show that CFIO improves the performance of climate models significantly versus the original serial I/O approach. When running with CFIO at 0.1° resolution with about 1000 CPU cores, we managed to reduce the running time by factors of 7.9, 4.6 and 2.0 for POP, CICE, and LICOM, respectively. We also compared the performance of CFIO against two existing libraries, PnetCDF and parallel I/O (PIO), in different scenarios. For scenarios with both data output and computations, CFIO decreases the I/O overhead compared to PnetCDF and PIO.

1 Introduction

Scientific computing for climate modeling has undergone radical changes over the past decade. One major trend is to increase the resolution of the models, so as to provide finer simulation of physical processes of the atmosphere, ocean, land, and sea ice. This trend is motivated by the availability of supercomputers with core counts in the range of tens to hundreds of thousands.

With a higher resolution, the amount of data generated by climate models will be significantly larger than before. In order to provide scientific data for the Fifth Assessment Report of the United Nations Intergovernmental Panel on Climate Change (IPCC AR5), modelers must run coupled climate models to simulate various types of climate change scenarios. The experiments in general last for months and generate hundreds of terabytes of data. The output of such a large amount of data results in severe performance degradation for numerical simulation experiments.

To improve the I/O performance of climate models, previous efforts and I/O libraries include a message passing interface-input/output (MPI-IO, Corbetty et al., 1996), the Network Common Data Form (netCDF, Rew and Davis, 1990), parallel netCDF (PnetCDF, Li et al., 2003), parallel I/O (PIO, Dennis et al., 2012), and adaptable I/O system (ADIOS, Lofstead et al., 2009, Lofstead et al., 2008).

Most of the above libraries attempt to improve the I/O throughput through parallelization techniques. For real applications, the overall running time mainly consists of two phases: computing time and I/O time. While the above libraries are helpful for shortening the I/O time of large-scale data, the computing phase still needs to wait for the I/O phase in iterative simulations. In a sense, the I/O phase and the computing phase are still serial with respect to each other. There is opportunity to improve I/O efficiency through overlapping the I/O phase and the computing phase.

With these issues in mind, we designed and implemented CFIO, a parallel I/O library that is specifically developed for climate models. The main idea of CFIO is to apply an I/O forwarding technique with client–sever architecture to provide automatic overlapping of I/O and computing. The strategy of overlapping I/O with computing as proposed in CFIO is complementary to existing parallel I/O libraries. Indeed,

Published by Copernicus Publications on behalf of the European Geosciences Union.

CFIO calls the PnetCDF functions directly to implement the parallel write and read on the CFIO server side. To minimize the code modifications required for porting, CFIO provides similar interfaces and features to PnetCDF, which is widely used by the climate community and different climate models.

We tested CFIO on three real-climate models: Parallel Ocean Program (POP, Smith et al., 2010), Community Ice CodE (CICE, Hunke and Lipscomb, 2010) and LASG/IAP climate system ocean model (LICOM, Yu et al., 2012). When running at 0.1° resolution with about 1000 CPU cores, we managed to decrease the running time by factors of 7.9, 4.6 and 2.0 for POP, CICE, and LICOM, respectively. We also compared the performance of CFIO against PnetCDF and PIO in different scenarios. Although CFIO has slightly lower throughput than PnetCDF and PIO for scenarios with only data output, CFIO decreases the I/O overhead compared to PnetCDF and PIO for scenarios with both data output and computations, resulting in better overall performance for real climate models.

The current release of CFIO is version 1.20. The source code and documentation for CFIO can be downloaded from the Github website (https://github.com/cfio/cfio).

The remainder of this paper is organized as follows. Section 2 discusses the motivation and the main idea of CFIO. The design and architecture of CFIO is presented in Sect. 3 in detail. Section 4 introduces the interface of CFIO and provides a simple example. Section 5 evaluates and analyses the performance of CFIO. Section 6 introduces related work. Conclusions and possible future work are discussed in Sect. 7.

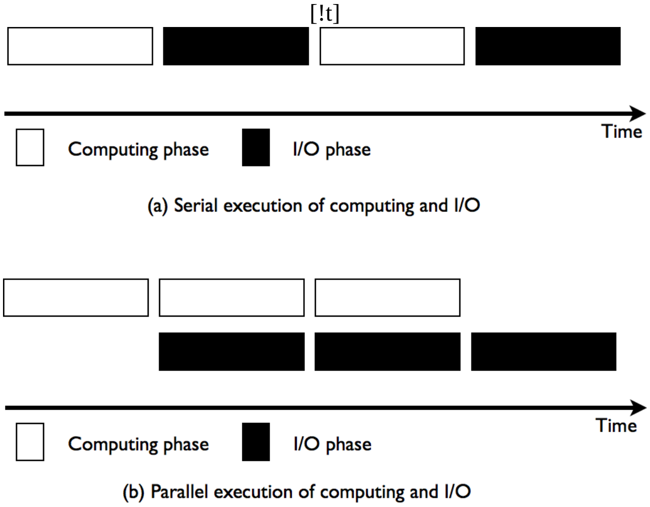
2 Motivation

In traditional climate models, the computing phase and the I/O phase run alternately. The computing phase performs simulation for certain time, and then the I/O phase outputs results following each computing phase. Namely, the computing phase and the I/O phase for traditional climate models are serial.

In fact, for most current climate models, the initial conditions or data sets for processing are loaded at the starting phase. Then the restart files, which contain all of the initial condition information that is necessary to restart from a previous simulation, will be written to the disk at a fixed frequency. Finally, the historic files, which include all of the diagnostic variables, will also be written to the disk at certain frequency. In general, there are no random seeks, no read-after-write operations, and writing operations are usually append only for all of the appropriate parts of the initial files, restart files and historic files.

Because of the append-only data accessing patterns, the computing step does not need to wait for the completion of the last I/O step. Motivated by this observation, we consider the possibility of overlapping the I/O phase with the computing phase. As shown in Fig. 1, the computing and the I/O phases can be performed in parallel. Compared with the serial computing and I/O method, the I/O time will be hidden by the computing time in a parallel computing and I/O method. This asynchronous computation and I/O will be useful to shorten the running time of climate models.

[!t]



(a) Serial execution of computing and I/O

(b) Parallel execution of computing and I/O

Fig. 1. Overlapping I/O with Computing.

Another advantage of overlapping the computing phase with the I/O phase is that the efficiency of computing and storage resources can be improved. For the serial method, the computing resource is idle during the I/O phase. On the other hand, the storage resource is idle during the computing phase. For the parallel method, the computing phase and the I/O phase are both pipelined. The computing and storage resources are always fully utilized.

3 Design of CFIO

This section describes the general design of CFIO. We first introduce the system architecture of CFIO and discuss the I/O forwarding technique, which is the main method to achieve the overlapping of the computing phase and the I/O phase. We then analyze the maximum possible speedup we can achieve by using CFIO. We also discuss the design options for synchronous and asynchronous communication methods of I/O forwarding.

3.1 System architecture of CFIO

Overlapping computation with communication and I/O is an established method for improving the performance of a parallel program. CFIO takes the advantage of the computing pattern to reduce the I/O overhead, and uses I/O forwarding to automate the overlapping of I/O and computing.

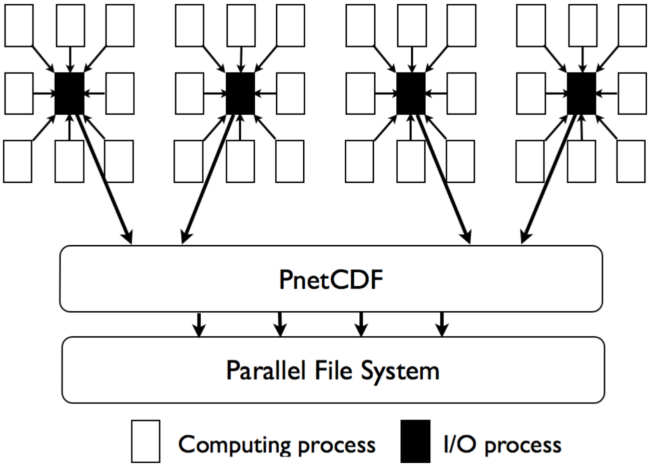


Fig. 2. The schematic diagram of the I/O forwarding technique.

A diagram of the I/O forwarding technique is shown in Fig. 2. The computing processes mainly handle the numerical computing tasks, while the I/O processes mainly handle the tasks of output data. All of the I/O requests generated in the computing processes will be forwarded to the I/O processes. The computing processes can therefore perform continuously without waiting for I/O to complete. The I/O processes output the data through calling the underlying PnetCDF interfaces. The I/O processes form a large buffer pool in memory and can be used to exchange data efficiently.

When the climate model uses CFIO as its I/O method, we would have a group of computing processes and an extra group of I/O processes to form the entire MPI communicator. For example, if we execute the original parallel program with 32 processes, and we want to use 4 CFIO processes to execute I/O operations, we will submit a parallel job with 36 processes.

The I/O forwarding technique has the following advantages: first, for each computing node, forwarding I/O requests to other nodes is useful for reducing the local competition for CPU and memory resources; second, the independent I/O processes provide a large memory buffer, which makes certain optimizations possible, such as data aggregation and rearrangement. The non-continuous writing of small data blocks can be transformed into continuous writing of large data blocks, which can significantly improve the performance of the parallel file system.

The system architecture of CFIO is shown in Fig. 3. We use a server–client mechanism to deal with forwarding and handling of I/O requests. The CFIO client is co-located with the computing process and provides the climate model with a series of interfaces for accessing the model data. When an I/O request is generated in a computing process, the CFIO client will pack the request into a message and then send it to the CFIO server via MPI communication.

[!t]

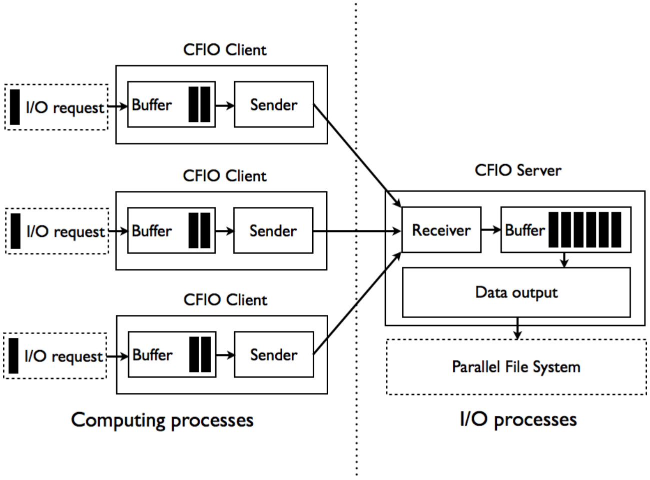


Fig. 3. The system architecture of CFIO.

The CFIO server is running as a daemon program in the I/O process to receive messages and handle I/O requests from multiple CFIO clients. After receiving an I/O request, the CFIO server places the request into an I/O queue. When handling I/O requests, the CFIO server takes one I/O request off the queue and unpacks the I/O request. The CFIO server then calls the corresponding PnetCDF function to perform the actual I/O operation. For data writing operations, data aggregation is performed to gather subarray data from each CFIO client into a large array of data.

For high-resolution climate models, we observe a significantly higher number of write operations than read operations. The initial conditions are always read only once and the results files are written many times. There is no space to overlap the read operation and computing. Thus the current CFIO v1.20 focuses on the write operations. All the parallel read operations in CFIO v1.20 will call the corresponding PnetCDF functions directly.

As shown in Fig. 4, the data is forwarded from the computing process to the I/O process. Thus, the total running time of the simulation consists of the computing time, the I/O forwarding time and the I/O time. One possible scenario that we must consider is that the I/O time is greater than the computing time, and can not be hidden by the computing phase. In this scenario, we can increase the number of I/O processes to solve the problem. More I/O processes provide a larger buffer pool, which can accommodate more data. Meanwhile, more I/O processes also lead to a faster writing speed. In this way, we can further reduce the I/O time and completely overlap the I/O phase with the computing phase.

3.2 Speedup analysis

In this section, we formulate an analytical model to estimate the maximum possible speedup of a program when switching to CFIO.

[!t]

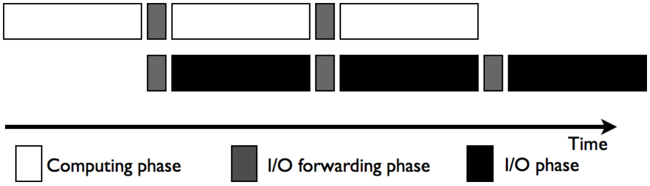


Fig. 4. Overlapping I/O with computing through I/O forwarding.

We denote the running time of a model with its default I/O approach and CFIO as T_{origin} and T_{cfio}, respectively. As shown in Fig. 4, T_{cfio} and T_{origin} can be calculated as follows:

$$T_{\mathrm{origin}} = T_{\mathrm{compute}} + T_{\mathrm{io}} \tag{1}$$

$$T_{\mathrm{cfio}} = \max\{T_{\mathrm{compute}} + T_{\mathrm{send}}, T_{\mathrm{recv}} + T_{\mathrm{io}}\}, \tag{2}$$

where T_{compute} and T_{io} are the computing time and the I/O time in one simulation step, T_{send} and T_{recv} are the time of sending I/O requests at the client and the time of receiving I/O requests at the server.

If the I/O time cannot be hidden by the computing phase, T_{cfio} equals the value of $(T_{\mathrm{recv}} + T_{\mathrm{io}})$. As mentioned above, this scenario can be avoided by increasing the number of I/O processes. So in an ideal case, $T_{\mathrm{cfio}} = T_{\mathrm{compute}} + T_{\mathrm{send}}$. The speedup S of using CFIO can be calculated as

$$S = \frac{T_{\mathrm{origin}}}{T_{\mathrm{cfio}}} = \frac{T_{\mathrm{compute}} + T_{\mathrm{io}}}{T_{\mathrm{compute}} + T_{\mathrm{send}}}. \tag{3}$$

An upper bound on speedup (when neglecting T_{send}) can be derived as

$$S < \frac{T_{\mathrm{compute}} + T_{\mathrm{io}}}{T_{\mathrm{compute}}} = 1 + \frac{T_{\mathrm{io}}}{T_{\mathrm{compute}}}. \tag{4}$$

The Eq. (4) means that the upper bound of speedup with CFIO is determined by the proportion of the I/O time and the computing time of the original program. The greater the proportion of I/O time in the entire running time, the greater speedup CFIO can achieve.

3.3 Communication method for I/O forwarding

In original program, the total running time of the simulation consists of the computing time and the I/O time. In the improved programs with CFIO, after the data has been forwarded from client to server, the computing phase and the I/O phase can be executed in parallel. So the ideal running time only includes the computing time and the I/O forwarding time.

Comparing the above two cases, we believe that the benefit of overlapping I/O with computing comes from the fact that the I/O forwarding time with a high speed network is much less than I/O time with a parallel file system, especially for the high-resolution climate models with a large amount of output data.

There are two options when designing the communication method for I/O forwarding: synchronous and asynchronous methods. The synchronous and asynchronous approaches we discussed here for data communication are only used to shorten the I/O forwarding time.

Our initial design for I/O forwarding is using the asynchronous communication approach. In this approach, all the I/O requests are packed into a client buffer; then, forwarding is performed by a separate sending thread during the computing phases. This approach permits I/O forwarding to overlap with computing, which implies that the major overhead of calling an asynchronous CFIO function is memory copy.

However, after performing many experiments, we observed that the asynchronous communication approach leads to network resource competition between the computing phase and the I/O forwarding phase. The competition overhead is negligible when we run the climate model with a small number of cores. However, when the number of cores increases to several hundreds, the competition leads to significant increase of the computing time, which completely overrides the benefits of overlapping the I/O forwarding with computing.

In contrast, the synchronous communication approach is a better choice for larger scale computing. When using this approach, the communication needed by the computing phase will not occur during the I/O forwarding phase. Therefore, the network resource competition is avoided. The effects of asynchronous communication and synchronous communication used in CFIO are compared in Sect. 5.1.

Note that synchronous communication can lead to buffer exhaustion in I/O processes because of the bursty I/O behavior in climate models. In this case, the CFIO client has to remain idle until the CFIO server finishes handling some of the buffered requests and releases sufficient buffer space. Because the I/O pattern of a climate model is known, the buffer exhaustion can be avoided by launching enough servers, the number of which is controlled by the user. How to determine the optimal number of CFIO servers for a particular program and a particular machine environment is the subject of ongoing work.

4 The CFIO interface

As the netCDF format is the de facto data format in the climate community, we choose to inherit the netCDF format to minimize the required efforts in terms of code updates and data post-processing when switching to CFIO. In netCDF, writing a new data set contains a sequence of operations, which creates the data set, defines the dimensions, variables, and attributes, ends define mode, writes variable data, and closes the data set file. CFIO supports all of the functions that are required to perform this series of operations. These functions can be classified into three categories:

Table 1. Additional Functions of CFIO.

Function	Description
cfio_init	Initialization of CFIO
cfio_final	Finalization of CFIO
cfio_proc_type	Indicating whether the process is an I/O process or a compute process
cfio_io_end	Indicating the completion of an I/O phase

1. *Data set Functions:* create/close a data set, set the data set to define/data mode.

2. *Define Mode Functions:* define data set dimensions, variables and attributes in define mode.

3. *Data Access Functions:* read/write variable data in data mode.

There are four additional functions that involve initialization and finalization of the library and an operation that relates to the I/O forwarding. All of the additional functions are shown in Table 1.

For the requirement of consistency across all computing processes, all CFIO functions are defined as collective I/O operations. For example, when a climate model intends to write a new data set, all of the computing processes should call the CFIO functions in the same sequence, and the same arguments should be passed into the functions.

Listing 1. A simple example with CFIO

```
!*******************************
! Initialize
!*******************************
ierr = cfio_init(LAT_PROC, LON_PROC, ratio)

!******************** ! computing and outputing
!********************
IF (cfio_proc_type() == CFIO_TYPE_CLIENT) THEN
    !********Computing phase************
    ......
    !********I/O phase*********************
    !output data
    ierr = cfio_create(filename, 0, ncid)
    ierr = cfio_def_dim(ncid,"lat",lat,dim(1))
    ierr = cfio_def_dim(ncid,"lon",lon,dim(2))
    ierr = cfio_def_var(ncid,"sst",NF_FLOAT,2,dim,0)
    ierr = cfio_enddef(ncid)
    ierr = cfio_put_vara_real(ncid,0,2,start,count,sst)
    ierr = cfio_close(ncid)
    ierr = cfio_end_io()
END IF

!*******************************
! Finalize
!*******************************
ierr = cfio_finalize()
!*******************************
```

Listing 1 shows a simple example of outputting data with CFIO. This example outputs sea surface temperature (SST) data with latitude and longitude dimensions. The *cfio_init* function is used to describe the dimensions of the output array and the number of CFIO servers. The function takes three arguments: *LAT_PROC*, *LON_PROC* and *ratio*. *LAT_PROC* and *LON_PROC* are used to describe the latitude and longitude dimension decompositions of the horizontal domain among computing processes; *ratio* stands for the proportion of computing processes and I/O processes. If we run the example application with N processes, there will be $N/ratio$ processes acting as I/O processes. The *cfio_proc_type* function is called to indicate the type of the local process. The computing processes should run the computing code and call the CFIO data access functions to output data. The I/O process will be launched automatically to run the CFIO server. The *cfio_put_vara_real* function is called to output the variable data and the *cfio_end_io* function is called to send a signal indicating that the current I/O phase is finished. This signal is used for the management of the buffer and the communication between the sender and the receiver.

This example shows that the I/O forwarding is automatically implemented and the complicated underlying mechanisms are opaque to the modelers. If the original program is already using netCDF interfaces, the users would not need to perform any extra programming rather than adding a few configuration function calls and switching "nf90" (the prefix for the standard netCDF Fortran 90 interfaces) into "cfio" when calling the I/O functions.

5 Experiments

We conducted our experiments on the *Tansuo100* supercomputer at Tsinghua University. The supercomputer consists of 740 nodes, each of which has two 2.93 GHz Intel Xeon X5670 6-core processors and 32 gigabytes (GB) memory. The nodes are connected through an Infiniband network, which provides a maximum bandwidth of 40 Gb s^{-1}. The file system is Lustre, with 1 Metadata Server (MDS) and 40 Object Storage Targets (OST). The peak writing performance of this file system is 4 GB s^{-1}. The node operating system is RedHat Enterprise Linux 5.5 ×86_64. All of the programs in our experiments are compiled with Intel compiler v11.1, and the MPI environment is Intel MPI v4.0.2.

In the following sections, we first describe our evaluation of CFIO through three climate models, and then provide a comparison between the performance of CFIO and PnetCDF in various scenarios. For the standalone POP, CICE and LICOM test cases, we downloaded the models from their official websites. The official standalone versions only support NetCDF.

In the following three cases, we provide the best performance of each model by turning off all I/O operations and comparing the result with CFIO. Because of the benefits brought by overlapping I/O with computing, using CFIO comes the closest to the best performance. Therefore, we believe our proposed forwarding scheme can be a useful complement to the existing parallel I/O libraries.

5.1 POP case study

POP is an ocean circulation model that has been developed at Los Alamos National Laboratory and written in Fortran with MPI support. We use version 2.0.1 of POP in this study. POP partitions the data arrays equally across all of the computing processes by using a two-dimensional data decomposition of the horizontal domain. In its default I/O approach, data outputs from different processes are gathered by one process and then written to disk by calling a serial netCDF interface. National Center for Atmospheric Research (NCAR) officially adopts POP as the ocean component of the Community Climate System Model (CCSM) and the Community Earth System Model (CESM), and has added various features to meet the needs of the CCSM and CESM coupled models. To enable experiments for high-resolution (0.1°) ocean modeling, we used the standalone POP instead of a coupled climate model.

The POP output files consist of restart files, history files and movie files. The restart file is generated for each simulated day. History and movie files are generated for each simulated hour. The variables included in the output netCDF files are two-dimensional arrays of 3600×2400, representing the spatial domain, and three-dimensional arrays of $3600 \times 2400 \times 40$, in which the third dimension represents sea depth. The total size of the final output files is 315 GB. In this experiment, the POP with 0.1° resolution ran for 440 iterations to simulate 2 days.

We recorded the overall POP running time with CFIO and compared the results with POP running with default I/O and with NO-I/O. The overall running time with NO-I/O describes the pure computation time, which can be used as the upper bound of the maximum performance that can be achieved by complete overlapping of I/O and computing. Figure 5 shows the experimental results. Our current design requires the number of computing processes to be multiples of the number of I/O processes. Therefore, the case of 64 I/O processes and 160 computing processes is not yet covered in our current experiments.

As expected, CFIO outperformed the default I/O approach in POP, with both 32 and 64 I/O processes. When running with 1280 computing processes and 32 I/O processes, the overall running time of POP decreased from 3246 s to 471 s, which means that we obtained a 6.9× speedup for POP with CFIO, which is already close to the upper bound given in the NO-I/O case. The performance of running with 64 I/O processes is better than that of running with 32 I/O processes. In the case of 1280 computing processes, we observed that the overall running time can be further reduced by 12 % (471 s down to 413 s) when switching from 32 to 64 I/O processes. This translates into an increased speedup of 7.9× when compared to the original performance of POP.

We also compared the POP running time for each of the two different communication approaches discussed in Sect. 3.3. To obtain an accurate understanding of the impact from I/O forwarding, we measured the computing time and I/O time separately in POP. We also evaluated the case with NO-I/O, to measure the pure computing running time that is not affected by any I/O operations.

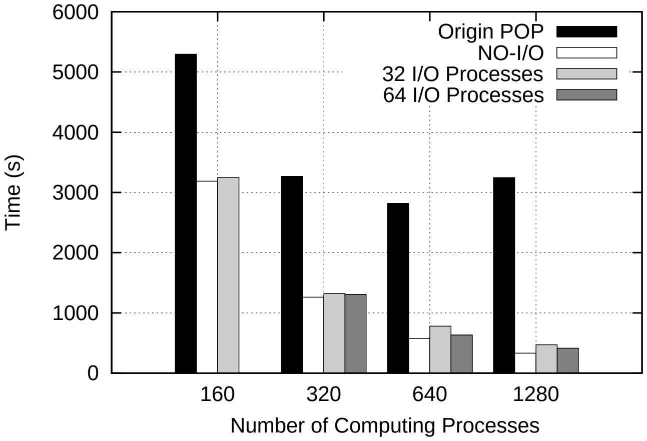


Fig. 5. The overall running time of POP with different I/O approaches.

Figure 6 shows the performance result for different communication methods when running with 32 I/O processes. We observed that when using the synchronous communication method, the computing time is always close to the case of NO-I/O. However, when using the asynchronous communication method, the computing time became significantly larger than the NO-I/O case when POP scales to a larger number of cores. POP running with 160 computing processes was the only case where the asynchronous communication method achieved shorter running time than the synchronous method. With the number of computing processes increased to 320, the total running time with the asynchronous communication method became larger than the synchronous method, due to the communication conflicts between the I/O forwarding and the computation. When running with 1280 computing processes, the computing time with the asynchronous communication method increased to 1101 s, which is around 3 times larger than the case with NO-I/O. Based on these results, we chose to use synchronous communication as our default communication method.

5.2 CICE case study

CICE is a sea ice model which has also been developed at Los Alamos National Laboratory. It is the sea ice component model of CESM. In general, CICE uses the same horizontal grid resolution as POP. CICE partitions the data arrays equally across all of the computing processes by using the same two-dimensional data decomposition as POP. CICE uses netCDF as the output file format for history files, and binary file format for the restart files. Similar to POP, when outputting the history file in CICE, data outputs are gathered by one process and then written to disk by calling a serial netCDF interface.

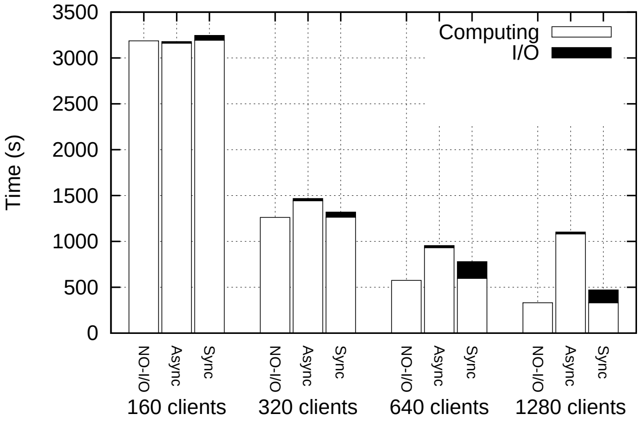


Fig. 6. The running time for computing and I/O with different communication methods when running with 32 CFIO servers.

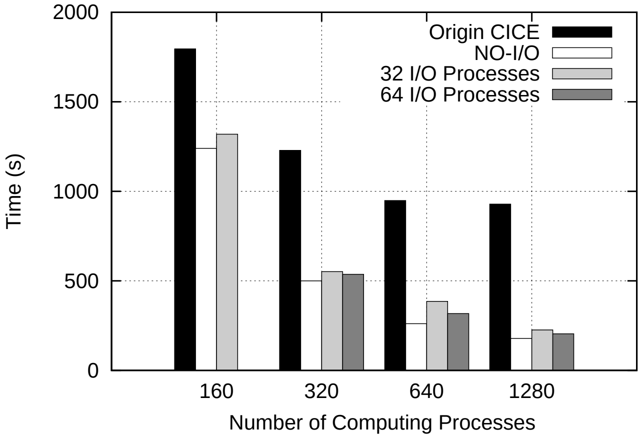


Fig. 7. The overall running time of CICE with different I/O approaches.

Because CFIO only supports netCDF format, we only used CFIO to output history files in CICE, and the output of restart files was disabled. We used the CICE version 4.1 with 0.1° resolution in this experiment. The CICE ran for 960 iterations to simulate 40 days. History files are generated for each simulated day. The variables included in output netCDF files are 2-D arrays of size 3600 × 2400. The output files have a fixed size of 80 GB in total.

We recorded the overall CICE running time with CFIO and compared the results with CICE running with default I/O and with NO-I/O. Figure 7 shows the experimental result. The number of computing processes varied from 160 to 1280. Comparing the running time of CICE with default I/O and NO-I/O, we clearly see that the I/O brings a significant overhead to both the running time and the scalability of the program. CFIO outperformed the default I/O approach in CICE, with both 32 and 64 I/O processes. When running with 1280 computing processes, the running time of CICE with 64 I/O processes was 204 s, which is less than the 226 seconds running time of CFIO with 32 CFIO servers. In terms of scalability, CICE with CFIO demonstrated a similar behavior to CICE with NO-I/O. Compared to CICE with default I/O (running time 928 s), we achieved 4.6× speedup by using 64 I/O processes.

The speedup in the case of CICE was slightly lower than the case of POP. The main reason is that the I/O load of CICE is not as heavy as POP. Since the running time of CICE with NO-I/O (the pure computing time) was 178 s, the proportion of the I/O time and the computing time is 4.2. Based on Eq. (4), we can infer that the maximum speedup from using CFIO in the CICE case is 5.2, compared with the maximum speedup of 9.8 in POP.

5.3 LICOM case study

LICOM is an ocean model developed by the State Key Laboratory of Numerical Modeling for Atmospheric Sciences and Geophysical Fluid Dynamics (LASG) of the Institute of Atmospheric Physics (IAP) in China. It is the sea component model of the LASG/IAP Earth System Model FGOALS_g2. LICOM also partitions the data arrays equally across all of the computing processes by using a two-dimensional data decomposition of the horizontal domain. LICOM uses netCDF as the output file format. The output data, including restart files and history files, are gathered by one process and then written to disk by calling a serial netCDF interface.

In this experiment, we used LICOM version 2 with 0.1° resolution to simulate 10 days. Restart files are generated for each simulated day. Only one restart file is generated at the end of the program. Output file variables are two-dimensional arrays of 3602 × 1683, in the spatial domain, and three-dimensional arrays of 3602 × 1683 × 55, in which the third dimension represents sea depth. The output files have a fixed size of 144 GB in total.

Figure 8 shows the test result of LICOM. In this experiment, the number of computing processes varied from 200 to 800. The scalability of LICOM is slightly poorer than POP and CICE. When scaling to 800 computing processes, the computing performance of LICOM started to degrade. Therefore, we used a maximum of 800 instead of 1280 computing processes in this experiment. LICOM running with 800 computing processes and 50 I/O processes had an running time of 4561 s. Compared to the original running time of 9101 s, LICOM obtained a 2 times speedup by using CFIO. The running time with NO-I/O (the pure computing time) was 4383 s, with the proportion of I/O time to computing time being 1.07. Based on Eq. (4), we can compute that the maximum speedup from using CFIO in the LICOM case is 2.07.

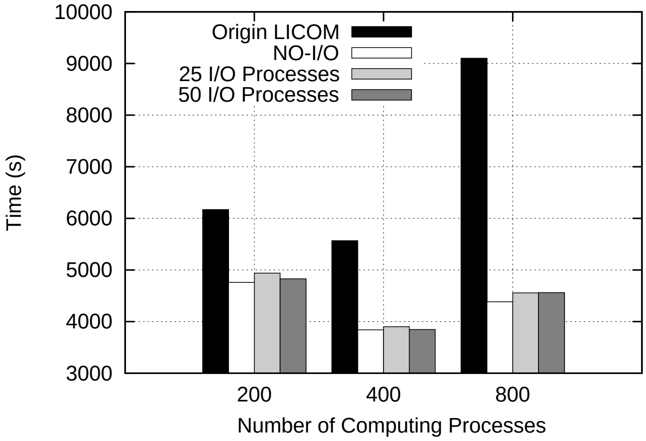


Fig. 8. The overall running time of LICOM with different I/O approaches.

5.4 Comparing CFIO with PnetCDF and PIO

As is well-known, PNetCDF was developed to support parallel I/O for NetCDF. It was based on MPI-IO to take advantage of collective I/O optimizations. PIO is an application-level parallel I/O library that was developed for the CESM. PIO supports several back-end I/O libraries, including MPI-IO, NetCDF, and PnetCDF.

In our experiments, we tested the newest PnetCDF v1.4.0 and PIO v1.6.0 and chose PnetCDF as the back-end method of PIO to obtain good parallel throughput. We refer interested readers to Dennis et al. (2012) for more detail about the performance of PnetCDF and PIO. The default striping count for our Lustre file system is 1, and we changed this argument to the maximum 40 to get the best write performance.

To compare the performance of CFIO, PnetCDF and PIO, we designed two MPI test programs to evaluate different scenarios. The first MPI test program outputs a 32 GB data set with 500 variables in one large netCDF file to evaluate the write bandwidth of CFIO, PnetCDF and PIO. The second MPI test program simulates the typical I/O patterns of climate models to show the advantage of I/O forwarding technology.

In the first program, every variable is a two-dimensional array with 4096 × 2048 double-precision floating-point numbers. Data arrays are partitioned equally across all of the computing processes, using two-dimensional data decomposition. The size of the output data per client process decreases as the number of computing processes increases.

Figure 9 shows the throughput of CFIO as a function of the number of CFIO servers. The throughput of PnetCDF and PIO are shown for comparison. The horizontal axis of PnetCDF and PIO stands for the number of clients that call PnetCDF functions. We see that the throughput of CFIO increased with the number of CFIO servers but then stopped increasing when the number of CFIO servers reached 128. This is mainly due to the limited number of storage devices in the Lustre file system. The writing throughput of CFIO reached approximately $1\,\mathrm{GB\,s^{-1}}$ when using 128 servers and 512 clients. The same pattern was observed for PnetCDF. PnetCDF achieved a throughput of approximately $1.24\,\mathrm{GB\,s^{-1}}$ when using 128 clients. The curve of PIO is very close to that of PnetCDF, peaking at $1.2\,\mathrm{GB\,s^{-1}}$ for 128 clients.

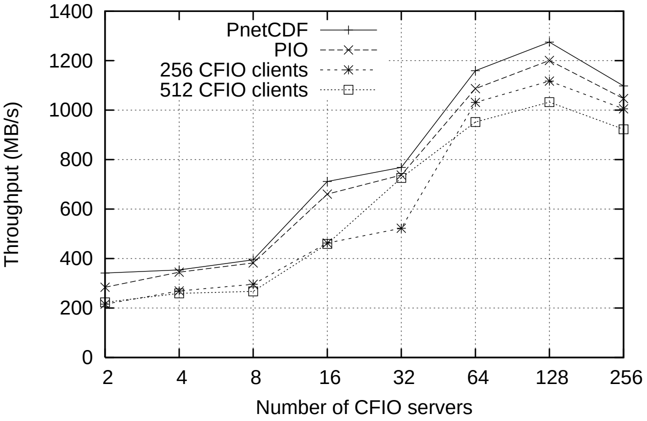


Fig. 9. Throughput comparison of CFIO, PnetCDF and PIO.

The results show that the throughput of CFIO is approximately 10% less than that of PnetCDF because of the overhead that is associated with I/O forwarding. Although CFIO provides slightly lower throughput than PnetCDF, we will show that the practical performance is better than PnetCDF in real scenarios emulated by the second program.

In the second program, we emulated a computing and I/O pattern that is typical in common climate models. There are a total of 40 loop iterations in the program. In each loop iteration, the program takes 7.5 s to perform floating-point computations, and will produce 3.2 GB of data. There are no intercommunication operations during the computing phases. For comparison purposes, we evaluated four different cases: CFIO, PnetCDF, PIO, and NO-I/O. NO-I/O means that all I/O operations are disabled in the second program.

Figure 10 shows the overall running time of the test program. Without any I/O operations, the total running time was 300 s. When running with 128 clients, the total running time using PnetCDF and PIO was 417 s and 431 s, respectively. PIO takes a small period of time, about 14 seconds, to initialize the I/O decomposition. The corresponding time using CFIO with 128 servers and 128 clients was 323 s. This result shows that CFIO decreases the I/O overhead compared to PnetCDF and PIO.

Figure 10 also shows that the performance of CFIO running with more servers is better. The I/O overhead with CFIO is the cost of I/O in the last loop, which cannot be overlapped, plus the cost of I/O forwarding. When the throughput of CFIO grows with the increase in the number of servers, the cost of I/O in the last loop is reduced. The cost of I/O forwarding is also naturally reduced as the number of CFIO servers increases.

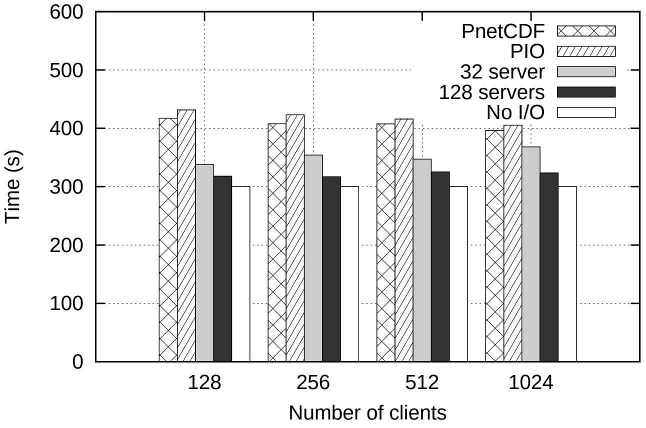


Fig. 10. The overall running time of the MPI test application with CFIO, PnetCDF and PIO.

6 Related work

Overlapping I/O has been shown as a useful technique to dramatically improve I/O performance. Dickens and Thakur (1999) implemented split-collective I/O in MPI-2 standard to provide a collective I/O that overlaps with computing by threads. The research found that simply spawning a thread to perform the collective I/O operation in the background is worse than the sequential approach. The best approach is to only perform writes to disk in the background, and to perform the copying and intercommunication that is required by the collective I/O in the main thread. More et al. (1997), Tsujita (2004) and Caglar et al. (2003) also used multi-threaded mechanisms to increase the performance of MPI applications. Patrick et al. (2008) presented a comparative study of different strategies of overlapping I/O, communication and computation. The performance results showed an obvious benefit from overlapping I/O with other procedures. Fu et al. (2010) quantified the overhead of overlapping I/O and computation using real partitioned Computational Fluid Dynamics(CFD) solver data. They proposed that it is possible to use a small portion (3 to 6 %) of the processes in the MPI communicator as I/O processes to achieve an actual writing bandwidth of 2.3 GB s^{-1} and latency hiding writing bandwidth of more than 21 TB s^{-1} on the IBM Blue Gene/L supercomputer.

Collective buffering is an attractive and practical optimization method in MPI-IO. It also called two-phase I/O, which means breaking the I/O operation into two stages. For a collective write operation, the first stage uses the aggregators, which are a subset of MPI processes, to aggregate the data into a temporary buffer on the aggregator nodes. In the second stage, the aggregators ship the data from the aggregator nodes to the I/O servers. The advantage of two-phase I/O is that fewer nodes are necessary to communicate with the I/O servers, which reduces resource contention. ROMIO (Thakur et al., 1999), which is one of the portable MPI-IO implementations, uses two-phase optimization to improve the I/O performance. ROMIO's two-phase optimization designates some MPI ranks to be the I/O aggregators, though ROMIO's aggregators are assigned to file regions, not to clients. The data model of ROMIO is a linear stream of bytes, whereas the data model of CFIO is array-oriented. It is worth noting that CFIO implements a form of the two-phase optimization implemented in ROMIO above the Lustre file system.

I/O forwarding has also been used in Prost et al. (2001), Oldfield et al. (2006), Nisar et al. (2008), Fu et al. (2010), Docan et al. (2010) and May (2001) to reduce the I/O impact on computing. The IBM Blue Gene series of supercomputers (Yu et al., 2006) uses independent I/O nodes in their system to handle I/O requests, which are generated in computer nodes and forwarded to I/O nodes. DataStager, designed by Abbasi et al. (2009), is a data staging service that provide asynchronous data extraction for ADIOS. ADIOS provides a simple function and an external XML file to configure the data structure and I/O methods. By switching parameters in the XML file, users can choose an optimal I/O method for their application according to the runtime environment. In ADIOS, a novel BP file format is designed to decrease the overhead of maintaining metadata consistency. DataStager uses server-directed I/O to manage asynchronous communication for data transfer. The DataStager research found that the asynchronous method for data transfer can significantly impact the performance of tightly coupled parallel programs. DataStager implements two schedulers to reduce the impact. The phase-aware scheduler prevents background data transfer in the communication phase, which is predicted by DataStager or marked by the application developers. The rate limiting scheduler manages the number of concurrent requests that are made to compute nodes to control the data transfer rate.

For climate models, Dennis et al. (2012) introduced an application level parallel I/O library named PIO. It provides the flexibility to adapt to different I/O requirements for different component models of CESM. PIO utilizes an I/O forwarding in which that a portion of the compute nodes are selected to collect the output data and rearrange data in memory into a more I/O-friendly decomposition, called data rearrangement. Through data rearrangement, PIO archives better I/O throughput with less memory consumption because it leads to less function calls of back-end I/O libraries. However, PIO cannot overlap I/O with computing, that is, PIO can shorten I/O time but not hide it. Palmer et al. (2011) also proposed a specialized parallel data I/O method for the Global Cloud Resolving Model. This method can avoid the creation of very large numbers of files. The output data layout linearizes the data in a consistent way that is independent of the number of processors used to run the simulation and provides a convenient format for subsequent analysis of the data.

The design of CFIO was inspired by the technologies of overlapping I/O, two-phase I/O and I/O forwarding which have been described above. Comparatively, CFIO focuses on the requirements of high-resolution climate models and provides automatic overlapping of I/O with computing so as to shorten the entire simulation time of the climate models, not just the I/O part. CFIO uses I/O forwarding to perform overlapping I/O on remote processors so that the overhead for managing multiple threads is avoided. In addition, CFIO provides synchronous functions that perform I/O overlapping automatically. Modifications to the existing climate modeling code for asynchronous functions are not necessary.

7 Conclusions

In this article, we presented a parallel I/O library, CFIO, which provides automated overlapping of I/O and computing. CFIO uses similar interfaces to PnetCDF, so as to minimize the required code modification when porting. The experimental results show that CFIO outperforms PnetCDF and PIO in typical climate modeling scenarios. We also compared the performance of using different communication methods, and we found that the synchronous communication method performs better when a program is running on a larger number of cores.

For future work, we plan to conduct more experiments on different machines with different file systems. We will also adopt and test CFIO in more climate models. The MPI communication method that we used for I/O forwarding requires further optimization. The method of determining the optimal number of CFIO servers for specific climate models still needs to be study.

Acknowledgements. This work is supported in part by a grant from the National Grand Fundamental Research 973 Program of China (no. 2014CB347800), the Natural Science Foundation of China (41375102, 61361120098) and the National High Technology Development Program of China (2011AA01A203). The authors gratefully acknowledge the valuable assistance of many people who commented on the design of CFIO. In particular, we thank Nick Clinton, Yanluan Lin, Fanghua Xu, and Peng Gong for their insightful comments on the preliminary draft of this paper. We also thank the editor and the reviewers who provided highly valuable and insightful comments for this paper.

Edited by: R. Redler

References

Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F.: DataStager: scalable data staging services for petascale applications, in: Proceedings of the 18th ACM international symposium on High performance distributed computing, 39–48, New York, NY, USA, 2009.

Caglar, S., Benson, G., Huang, Q., and Chu, C.: USFMPI: A multi-threaded implementation of MPI for Linux clusters, in: Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems, 2003.

Corbetty, P., Feitelsony, D., Fineberg, S., Hsuy, Y., Nitzberg, B., Prosty, J., Sniry, M., Traversat, B., and Wong, P.: Overview of the MPI-IO parallel I/O interface, Input/output in parallel and distributed computer systems, 362, 127–146, 1996.

Dennis, J., Edwards, J., Loy, R., Jacob, R., Mirin, A., Craig, A., and Vertenstein, M.: An application-level parallel I/O library for Earth system models, Int. J. High Performance Comput. Appl., 26, 43–53, 2012.

Dickens, P. and Thakur, R.: Improving collective I/O performance using threads, in: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, 38–45, Washington, DC, USA, 1999.

Docan, C., Parashar, M., and Klasky, S.: Enabling high-speed asynchronous data extraction and transfer using DART, Concurr. Comp. Pract. E., 22, 1181–1204, 2010.

Fu, J., Liu, N., Sahni, O., Jansen, K. E., Shephard, M. S., and Carothers, C. D.: Scalable parallel I/O alternatives for massively parallel partitioned solver systems, in: Proceedings of the Workshop on Large-Scale Parallel Processing in conjunction with the IEEE International Parallel and Distributed Processing Symposium, 1–8, Atlanta, Georgia, USA, 2010.

Hunke, E. and Lipscomb, W.: CICE: the Los Alamos Sea Ice Model documentation and software user's manual version 4.1, Rep. LA-CC-06, 12, 2010.

Li, J., Liao, W.-K., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M.: Parallel netCDF: A High-Performance Scientific I/O Interface, in: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, 39–39, New York, NY, USA, 2003.

Lofstead, J., Zheng, F., Klasky, S., and Schwan, K.: Input/output apis and data organization for high performance scientific computing, in: Proceedings of Petascale Data Storage Workshop 2008 at Supercomputing 2008, 1–6, Austin, Texas, USA, 2008.

Lofstead, J., Zheng, F., Klasky, S., and Schwan, K.: Adaptable, metadata rich IO methods for portable high performance IO, in: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, 1–10, Washington, DC, USA, 2009.

May, J. M.: Parallel I/O for high performance computing, Morgan Kaufmann, 2001.

More, S., Choudhary, A., Foster, I., and Xu, M.: MTIO – A Multi-Threaded Parallel I/O System, in: Proceedings of the 11th International Symposium on Parallel Processing, 368–373, Washington, DC, USA, 1997.

Nisar, A., Liao, W.-k., and Choudhary, A.: Scaling parallel I/O performance through I/O delegate and caching system, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 1–12, Piscataway, NJ, USA, 2008.

Oldfield, R., Ward, L., Riesen, R., Maccabe, A., Widener, P., and Kordenbrock, T.: Lightweight I/O for scientific applications, in: Proceedings of the IEEE International Conference on Cluster Computing, 1–11, Barcelona, Spain, 2006.

Palmer, B., Koontz, A., Schuchardt, K., Heikes, R., and Randall, D.: Efficient data IO for a parallel global cloud resolving model, Environ. Model. Softw., 26, 1725–1735, 2011.

Patrick, C., Son, S., and Kandemir, M.: Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO, ACM SIGOPS Operating Systems Review, 42, 43–49, 2008.

Prost, J.-P., Treumann, R., Hedges, R., Jia, B., and Koniges, A.: MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS, in: Proceedings of the 2001 ACM/IEEE conference on Supercomputing, p. 58, New York, NY, USA, 2001.

Rew, R. and Davis, G.: NetCDF: an interface for scientific data access, IEEE Comput. Graphics Appl., 10, 76–82, 1990.

Smith, R., Jones, P., Briegleb, B., Bryan, F., Danabasoglu, G., Dennis, J., Dukowicz, J., Eden, C., Fox-Kemper, B., Gent, P., Hecht, M., Jayne, S., Jochum, M., Large, W., Lindsay, K., Maltrud, M., Norton, N., Peacock, S., Vertenstein, M., and Yeager, S.: The Parallel Ocean Program (POP) Reference Manual: Ocean Component of the Community Climate System Model (CCSM) and Community Earth System Model (CESM), Tech. Rep. LAUR-10-01853, Los Alamos National Laboratory, 2010.

Thakur, R., Gropp, W., and Lusk, E.: Data Sieving and Collective I/O in ROMIO, in: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation, 182–189, Washington, DC, USA, 1999.

Tsujita, Y.: Effective nonblocking MPI-I/O in remote I/O operations using a multithreaded mechanism, in: Proceedings of the Second international conference on Parallel and Distributed Processing and Applications, 34–43, Berlin, Heidelberg, 2004.

Yu, H., Sahoo, R. K., Howson, C., Almasi, G., Castaños, J. G., Gupta, M., Moreira, J. E., Parker, J. J., Engelsiepen, T., Ross, R. B., Thakur, R., Latham, R., and Gropp, W. D.: High performance file I/O for the Blue Gene/L supercomputer, in: International Symposium on High-Performance Computer Architecture, 187–196, Austin, Texas, USA, 2006.

Yu, Y., Liu, H., and Lin, P.: A quasi-global 1/10° eddy-resolving ocean general circulation model and its preliminary results, Chinese Sci. Bull., 57, 3908–3916, 2012.