DTF: An I/O Arbitration Framework for Multi-component Data Processing Workflows

Tatiana V. Martsinkevich[1](✉), Balazs Gerofi[1], Guo-Yuan Lien[1], Seiya Nishizawa[1], Wei-keng Liao[2], Takemasa Miyoshi[1], Hirofumi Tomita[1], Yutaka Ishikawa[1], and Alok Choudhary[2]

[1] RIKEN AICS, Tokyo, Japan
tatiana.mar@riken.jp
[2] Northwestern University, Chicago, USA

Abstract. Multi-component workflows, where one component performs a particular transformation with the data and passes it on to the next component, is a common way of performing complex computations. Using components as building blocks we can apply sophisticated data processing algorithms to large volumes of data. Because the components may be developed independently, they often use file I/O and the Parallel File System to pass data. However, as the data volume increases, file I/O quickly becomes the bottleneck in such workflows. In this work, we propose an I/O arbitration framework called DTF to alleviate this problem by silently replacing file I/O with direct data transfer between the components. DTF treats file I/O calls as I/O requests and performs I/O request matching to perform data movement. Currently, the framework works with PnetCDF-based multi-component workflows. It requires minimal modifications to applications and allows the user to easily control I/O flow via the framework's configuration file.

Keywords: Multi-component workflow · Workflow coupling
I/O performance · I/O arbitration

1 Introduction

In the past several years, the steady growth of computational power that newly built High Performance Computing (HPC) systems can deliver allowed humanity to tackle more complex scientific problems and advance data-driven sciences.

Rather than using the conventional monolithic design, many applications running in these systems are multi-component workflows in which components work together to achieve a common goal. Each component performs a particular task, such as building different physics models or running a model with different parameters as it is done in ensemble simulation programs. The result of the computation is then passed to the next component for further processing. A workflow may also include components for data analytics, in-situ visualization and so on. Components may be either loosely coupled, i.e., by using files to pass data, or they can use a coupling software.

Such work pipelining allows to build powerful complex programs that perform sophisticated data processing. The module-based approach can also facilitate the development of new programs as they can be built fast by combining components from previously developed workflows. Many recent research works focus on tuning HPC systems so that they could run multi-component workflows more efficiently [1,2]. The I/O bottleneck in such applications is one of the issues that receives a lot of attention. The faster the component receives the data from the previous component, the sooner it can start processing it. However, for workflows coupled through files, file I/O can become a bottleneck, especially when they pass large amounts of data.

A motivating real-world application example for this work is an application called SCALE-LETKF [3]. SCALE-LETKF is a real-time severe weather prediction application that combines weather simulation with assimilation of weather radar observations. It consists of two components (Fig. 1) — SCALE and LETKF — that are developed independently. SCALE is a numerical weather prediction application based on the ensemble simulation; LETKF performs data assimilation of real-world observation data together with simulation results produced by SCALE.

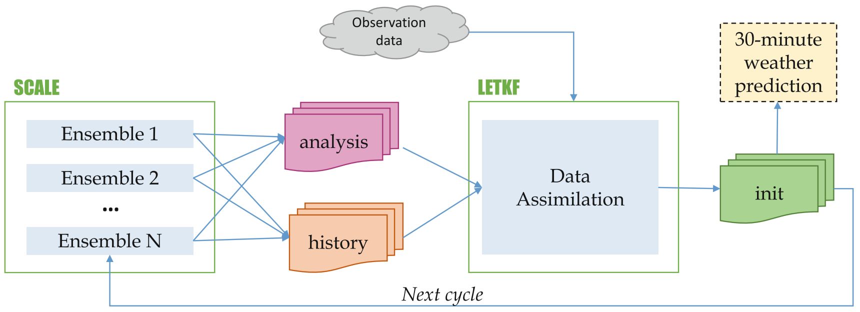


Fig. 1. SCALE-LETKF

In each iteration, SCALE writes the simulation result to the Parallel File System (PFS) using the Parallel NetCDF [4] API. The files are subsequently read by LETKF. After LETKF finishes assimilating the observation data, the output is written to files which become the input for SCALE in the next iteration. One of the results from every iteration is also used to predict weather for the next 30 min on separate compute nodes.

A particular feature of SCALE-LETKF is that it has a strict timeliness requirement. The target execution scenario is to assimilate the observations arriving at an interval of 30 s. Therefore, one full iteration of SCALE-LETKF, including the computations and I/O, must finish within this time period. However, this requirement quickly becomes hard to fulfill once the amount of file I/O grows too big.

One way to overcome this would be to switch from file I/O to some coupling software. However, this would require rewriting the I/O kernels in both components using the API of the coupler as such a software usually requires. This can be a daunting task for a software as large and complex as SCALE-LETKF.

In this work, we propose a framework called Data Transfer Framework (DTF) that silently bypasses file I/O by sending the data directly over network and requires minimal modifications to application source code. Current implementation of DTF assumes that the workflow components use PnetCDF API for file I/O. The framework uses the Message Passing Interface (MPI) [5] library to transfer the data. Applications like SCALE-LETKF can benefit from DTF because it allows the developers of the application to easily switch from file I/O to direct data transfer without having to rewrite the I/O code.

The main contributions of this work are:

- We propose a simple data transfer framework called DTF that can be used to silently replace PnetCDF-based file I/O with direct data transfer;
- Unlike many existing coupling solutions, DTF requires only minimal modifications to the application and does not require modifications of PnetCDF calls themselves;
- DTF automatically detects what data should be transferred to what processes transparently for the user, hence, it can be plugged into workflows fast and with minimal efforts and there is no need to provide a description of I/O patterns across components;
- Using a benchmark program, we show that the DTF exhibits stable performance under different I/O loads. A test run of DTF with the real-world workflow application (SCALE-LETKF) shows that the DTF can help multi-component workflows achieve real-time requirements.

The rest of this paper is organized as follows. In Sect. 2 we present in detail the design of our data transfer framework and discuss its implementation in Sect. 3. We present the results of the performance evaluation of the framework in Sect. 4. In Sect. 5 we overview existing solutions proposed to facilitate the data movement between the components in multi-component workflows. Finally, we conclude with Sect. 6.

2 Data Transfer Framework

DTF can be used in workflows in which the components use the PnetCDF library for file I/O. In this section, we first present some basic concepts of the (P)netCDF data format that had a direct influence on the design of the DTF. We then present the general overview of the framework and, finally, discuss in detail how the data transfer is performed.

We note that from now on we will call the component that writes the file and the component that reads it as the writer and the reader components, respectively.

2.1 Parallel NetCDF Semantics

Network Common Data Form [6] is a self-describing portable data format that supports handling of array-oriented scientific data. The NetCDF library provides users with an API that allows them to create files conforming to this data format and to define, store and retrieve the data. Parallel NetCDF (PnetCDF) is, as the name suggests, a parallel implementation of the NetCDF library. PnetCDF utilizes the parallel MPI-IO under the hood which allows multiple processes to share the file.

Before performing I/O, the user must first define the structure of the file, that is, define variables, their attributes, variable dimensions and dimension lengths.

Once the structure of the file is defined, the user may call PnetCDF's API to read or write variables. In a typical PnetCDF call, the user must specify the file id and variable id, which were assigned by PnetCDF during the definition phase, specify the start coordinate and block size in each dimension for multi-dimensional variables, and pass the input or output user buffer.

Similarly to MPI, PnetCDF has blocking and non-blocking API. In non-blocking I/O, the user first posts I/O requests and then calls a wait function to force the actual I/O. The purpose of non-blocking calls is to allow processes to aggregate several smaller file I/O requests into a larger request to improve the I/O throughput.

2.2 General Overview of DTF

DTF aims to provide users of multi-component workflows with a tool that would allow them to quickly switch from file I/O to direct data transfer without needing to cardinally change the source code of the components.

First, the user must provide a simple configuration file that describes the file dependency in the workflow (example in Fig. 4). It only needs to list the files that create a direct dependency between two components, i.e. if the components are coupled through this file. The DTF intercepts PnetCDF calls in the program and, if the file for which the call was made is listed in the configuration file as subject to the data transfer, the DTF handles the call accordingly. Otherwise, PnetCDF call is executed normally.

In order to transfer the data from one component to another, we treat every PnetCDF read or write call as an *I/O request*. The data transfer is performed via what we call *the I/O request matching*. First, designated processes, called *I/O request matchers*, collect all read and write requests for a given file. Then, each matcher finds out who holds the requested piece of data by matching each read request against one or several write requests. Finally, the matcher instructs the processes who have the data to send it to the corresponding process who requested it. All the inter-process communication happens using MPI. We note that here we differentiate between the PnetCDF non-blocking I/O requests and the DTF I/O requests, and we always assume the latter unless stated otherwise.

The I/O patterns of the component that writes to the file and the component that reads from it may be drastically different, however, dynamic I/O request matching makes DTF flexible and allows it to handle any kind of I/O patterns transparently for the user.

2.3 I/O Request Matching

When the writer component creates a file, matchers that will be handling the I/O request matching are chosen among its processes. The number of matchers can be configured by the user or else a default value is set.

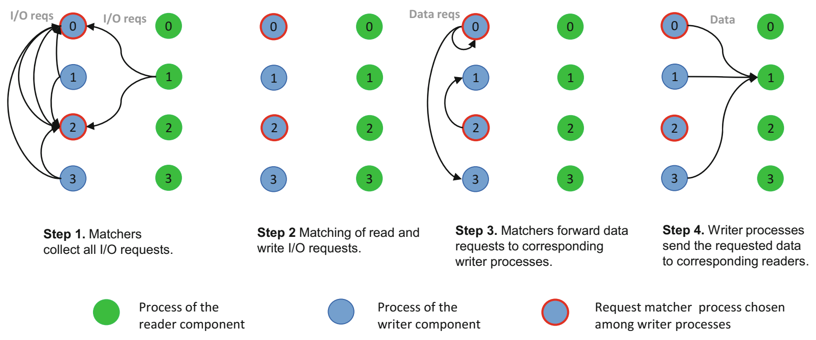

I/O reqs
I/O reqs
Data reqs
Data
0
1
2
3
Step 1. Matchers collect all I/O requests.
Step 2 Matching of read and write I/O requests.
Step 3. Matchers forward data requests to corresponding writer processes.
Step 4. Writer processes send the requested data to corresponding readers.
Process of the reader component
Process of the writer component
Request matcher process chosen among writer processes

Fig. 2. I/O request matching. Request matchers are marked with a red shape outline. For simplicity, only one reader process is showed to have read I/O requests. (Color figure online)

When a process calls a read or write PnetCDF function for a file intended for data transfer, the DTF intercepts this call and, instead of performing file I/O, it internally creates an I/O request that stores the call metadata. Additionally, the process may buffer the user data if the DTF is configured to do so. The metadata consists of:

- `varid` - the PnetCDF variable id;
- `rw flag` - read or write request;
- `datatype` - in case this datatype does not match with the datatype used when the variable was defined, type conversion will take place;
- `start` - corner coordinate of the array block;
- `count` - length of the block in each dimension;
- `buffer` - pointer to the user buffer.

The request matching process can be divided in four steps (Fig. 2). First, all the processes of the reader and writer component send all their I/O requests posted so far to corresponding matching processes (Step 1). Then, a matching process takes the next read I/O request and, based on the corner coordinate `start` of the requested array block and the block size `count`, searches for matching write requests (Step 2). The I/O pattern of the reader and writer components are not necessarily identical, therefore, one read request may be matched with several write requests, each of them - for a sub-block of the requested array block. Once a match is found, the matcher sends a message to the writer process holding the requested portion of data and asks it to send this data to the corresponding reader process (Step 3). Finally, when a writer process receives a data request from the matcher, it finds the requested data in the memory, copies it to the send buffer along with the metadata and sends it to the reader (Step 4). When the reader receives the message, it parses the metadata and unpacks the data to the user buffer.

For better performance, the requests are distributed among the matching processes and each matcher is in charge of matching requests for a particular sub-block of a multi-dimensional variable. The size of the sub-block is determined by dividing the length of the variable in the lowest (zeroth) dimension by the number of matching processes. If there is a request that overlaps blocks handled by different matchers, such a request will be split into several requests for sub-blocks, and each matcher will match the corresponding part. There is a trade-off in this approach: On one hand, the matching happens in a distributed fashion, on the other hand, if there are too many matchers the request may end up being split too many times resulting in more communication between readers and writers. Therefore, it is recommended to do some test runs of the workflow with different number of matchers to find a reasonable configuration for DTF.

3 Implementation

The Data Transfer Framework is implemented as a library providing API to user programs. To let the DTF intercept PnetCDF calls, we also modified the PnetCDF-1.7.0 library. The modifications were relatively small and consisted of around 50 lines of code.

We use the MPI library to transfer the data. To establish the communication between processes in the reader and writer components, we use the standard MPI API for creating an inter-communicator during the DTF initialization stage in both components. This implies that the two components coupled through a file must run concurrently.

Current version of the DTF implements a synchronous data transfer, meaning that all the processes in the two components stop their computations to perform the data transfer and resume only when all the data requested by the reader has been received. Generally, it is preferable to transfer the data to the reader as soon as it becomes available on the writer's side so that the reader could proceed with computations. However, because the I/O patterns of the two components may differ significantly, it is hard to automatically determine when it is safe to start the matching process. Therefore, we require that the user signals to the DTF when to perform a request matching for a given file by explicitly invoking a special API function in both components.

To enable the data transfer, the user needs to modify the source code of all the components of the workflow by adding API to initialize, finalize the DTF, as well as explicitly invoke the data transfer. However, we believe that

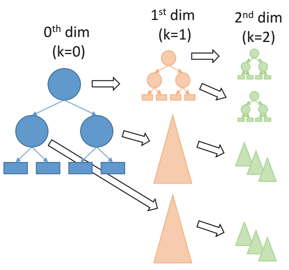
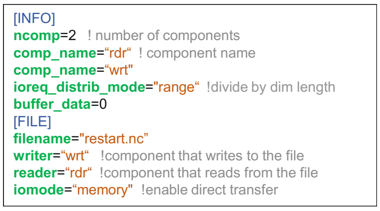


Fig. 3. An example layout of a k-d tree to arrange sub-blocks of a 3-dimensional variable.



```
[INFO]
ncomp=2   ! number of components
comp_name="rdr"  ! component name
comp_name="wrt"
ioreq_distrib_mode="range"  !divide by dim length
buffer_data=0
[FILE]
filename="restart.nc"
writer="wrt"   !component that writes to the file
reader="rdr"   !component that reads from the file
iomode="memory"  !enable direct transfer
```

Fig. 4. DTF configuration file

these modifications are rather minimal compared to what traditional coupling software usually requires.

3.1 Handling of I/O Requests

Depending on the scale of the execution and the I/O pattern, matching processes sometimes may have to handle thousands of I/O requests. Using a suitable data structure to arrange the requests in such a way that matching read and write requests can be found fast is important.

Unless the variable is a scalar, an I/O request is issued for a multi-dimensional block of data. Such `k`-dimensional block can be represented as a set of `k` intervals. We use an augmented `k`-dimensional interval tree [7] to arrange these blocks in such a way that would allow us to find a block that overlaps with a quired block (read I/O request) in a reasonable amount of time. Figure 3 shows an example layout of a tree that stores write requests for a 3-dimensional variable. A tree on each level ($\mathtt{k}$ = 0,1,2) arranges intervals of the variable sub-blocks for which a write request was issued in the corresponding dimension. Each node of the tree links to the tree in the $\mathtt{k} + 1$ dimension.

Read requests are stored as a linked list sorted by the rank of the reader. Every time new requests metadata arrives, the matcher updates the request database and tries to match as many read requests for a given rank as possible.

3.2 User API

The three main API functions provided by the DTF are the following:

- `dtf_init(config_file, component_name)` - initializes the DTF. The user must specify the path to the DTF configuration file and state the name of the current component which should match one of the component names in the configuration file;
- `dtf_finalize()` - finalizes the DTF;
- `dtf_transfer(filename)` - invokes the data transfer for file *filename*;

All the API functions are collective: `dtf_init()` and `dtf_finalize()` must be invoked by all processes in both components, while `dtf_transfer()` must be invoked only by processes that share the file.

During the initialization, based on the DTF configuration file, each component finds out all other components with whom it has an I/O dependency and establishes a separate MPI inter-communicator for every such dependency. All the further inter-component communication happens via this inter-communicator.

A `dtf_transfer()` call should be added after corresponding PnetCDF read-/write calls in the source code of both, reader and writer components. The call will not complete until the reader receives the data for all the read I/O requests posted before `dtf_transfer()` was invoked, therefore, it is user's responsibility to ensure that the components call the function in the correct place in the code, that is, that the writer does not start matching I/O until all the write calls for the data that will be requested in the current transfer phase have been posted as well. `dtf_transfer()` function can be invoked arbitrary number of times but this number should be the same for both components. We note that, because this function acts like a synchronizer between the reader and writer components, the recommended practice is to invoke it just once after all the I/O calls and before the file is closed.

By default, the DTF does not buffer the user data internally. Therefore, the user should ensure that the content of the user buffer is not modified between the moment the write PnetCDF call was made until the moment the data transfer starts. Otherwise, data buffering can be enabled in the DTF configuration file. In this case, all the data buffered on the writer's side will be deleted when a corresponding transfer function is completed.

3.3 Example Program

A simplified example of a writer and reader components is presented Figs. 5a and b, as well as their common DTF configuration file (Fig. 4). To enable the direct data transfer it was enough to add three lines of code to each component — to initialize, finalize the library and to invoke the data transfer — and provide a simple configuration file.

4 Evaluation

We first demonstrate the performance of DTF using the S3D-IO[1] benchmark program. Next, we show how the DTF performs with a real world workflow application—SCALE-LETKF.

S3D-IO [8] is the I/O kernel of the S3D combustion simulation code developed at Sandia National Laboratory. In the benchmark, a checkpoint file is written at regular intervals. The checkpoint consists of four variables—two three-dimensional and two four-dimensional—representing mass, velocity, pressure,

[1] Available at http://cucis.ece.northwestern.edu/projects/PnetCDF/#Benchmarks.

```
/*Initialize DTF*/
dtf_init(dtf_inifile, "wrt");
/*Create file*/
ncmpi_create("restart.nc",...);
<...>
 /*Write some data*/
ncmpi_put_vara_float(...);
/*Write some more data*/
ncmpi_put_vara_float(...);
/*Perform I/O request matching*/
dtf_transfer("restart.nc");
/*Close the file*/
ncmpi_close(...);
/*Finalize DTF*/
dtf_finalize();
```

(a) Component writing to file

```
/*Initialize DTF*/
dtf_init(dtf_inifile, "rdr");
/*Open the file*/
ncmpi_open("restart.nc",...);
<...>
/*Read all data at once*/
ncmpi_get_vara_float(...);
/*Perform I/O request matching*/
dtf_transfer("restart.nc");
/*Close the file*/
ncmpi_close(...);
/*Finalize DTF*/
dtf_finalize();
```

(b) Component reading from file

Fig. 5. Sample code using the DTF API

and temperature. All four variables share the lowest three spatial dimensions X, Y and Z which are partitioned among the processes in block fashion. The value of the fourth dimension is fixed.

We imitate a multi-component execution in S3D-IO by running concurrently two instances of the benchmark: Processes of the first instance write to a shared file, processes in the second instance read from it. Each test is executed at least eight times and an average value of the measured parameter is computed. To determine the number of matchers necessary to get the best performance for data transfer, we first execute several tests of S3D-IO varying the number of matching processes and use the result in the subsequent tests.

In the tests with the direct data transfer, the I/O time was measured in the following manner. On the reader side, it is the time from the moment the reader calls the data transfer function to the moment all its processes received all the data they had requested. On the writer's side, the I/O time is the time between the data transfer function and the moment the writer receives a notification from the reader indicating that it had got all the requested data. The I/O time also includes the time to register the metadata of the PnetCDF I/O calls and to buffer the data, if this option is enabled. In all our test cases it so happens that the writer component always invokes the transfer function before the reader and, therefore, sometimes it has to wait for the reader to catch up. Hence, by data transfer time we hereafter assume the I/O time of the writer component unless stated otherwise as it represents the lowest baseline. The runtime of the workflow is measured from the moment two components create an MPI inter-communicator inside the `dtf_init()` function and the moment it is destroyed in `dtf_finalize()` as these two functions work as a synchronization mechanism between the reader and writer components.

All the experiments were executed on K computer [9]. Each node has an 8-core 2.0 GHz SPARC64 VIIIfx CPU equipped with 16 GB of memory. Nodes are connected by a 6D mesh/torus network called Tofu [10] with 5 GB/s $\times$ 2 bandwidth in each link. Compute nodes in K computer have access to a local per-node file system as well as a global shared file system based on Lustre file system.

4.1 S3D-IO Benchmark

Choosing the Number of Matching Processes. To get the best performance, it is recommended that the user chooses the number of matching processes that will perform I/O matching instead of using the default configuration of one matcher per 64 processes. This number is application-dependent. The load on a matching process is determined by the number of read and write I/O requests the process has to match. For example, if all reader and writer processes perform I/O symmetrically and the size of the variable in the zeroth dimension divides by the number of matchers, the number of I/O requests one matcher will have to match roughly equals the number of I/O requests one process generates multiplied by the number of processes in both components.

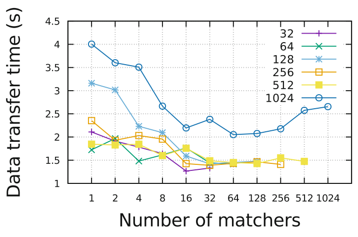


Fig. 6. Data transfer time for various test sizes and number of matching processes per component.

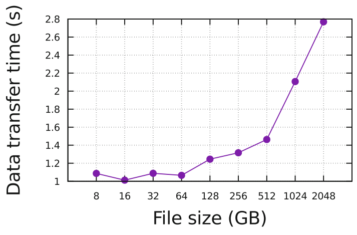


Fig. 7. DTF performance for various file sizes.

Depending on the I/O pattern, increasing the number of matchers does not always decrease the number of I/O requests per matcher, but it generally improves the throughput of data transfer. The reason is that rather than waiting for one matching process to match requests for one block of a multi-dimensional array, multiple processes can match sub-blocks of it in parallel. Consequently, the reader may start receiving the data earlier.

To find an optimal number of matchers, we run tests of different sizes—from 32 processes per component up to 1024—with a problem size such that each process reads or writes 1 GB of data. In each test we then vary the number of matchers and measure the time to transfer the data. The results in Fig. 6 show that increasing the number of matchers up to some point improves the transfer time and then the performance starts decreasing. The reason for this is that an I/O request for a block of data may be split into several requests for sub-blocks between multiple matchers and, if the number of matchers is too big, the request is over-split and it takes more smaller messages to deliver all the data to the reader. Based on this result, for our further tests we use the following setting: for tests with up to 256 processes in one component, each writer process functions as a matcher, for tests with 512 processes per component—four processes in one work-group, i.e. 128 matchers in total. Finally, for tests with 1024 processes per component the work-group size is 16, i.e. there are 64 matchers in total.

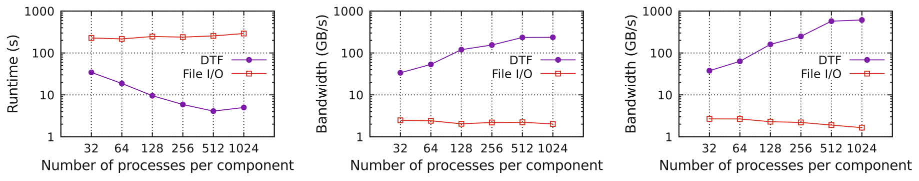
Scalability. We first demonstrate how the DTF scales compared to file I/O (PnetCDF) by measuring the read and write bandwidth for weak and strong scaling tests. In this test, processes write to a shared file using non-blocking PnetCDF calls. To measure the I/O bandwidth, we divide the total file size by the respective read or write I/O time. The results for the strong and weak scaling are presented on Figs. 8 and 9. The X axis denotes the number of processes in one component. We point out that the Y-axis is logarithmic in these plots. Figures 8a and 9a show the total execution time of the coupled workflow.

In all tests each process executes a PnetCDF read or write function four times—one per variable, i.e. each process generates four I/O requests.

In the strong scaling test, we fix the file size to 256 GB and vary the number of processes in one component. We note that, due to the node memory in K computer limited to 16 GB, the results in Fig. 8 start from the test with 32 processes per component. In the weak scaling tests, we fix the size of the data written or read by one process to 256 MB, thus, in the test with one process per component the file size is 256 MB, in the test with 1024 processes—256 GB.

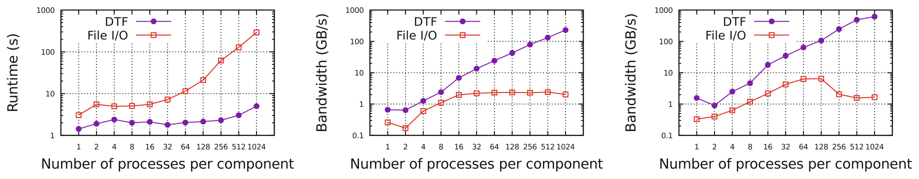
As we see, DTF significantly outperforms file I/O in all tests. We also notice that the read bandwidth in all tests with the direct data transfer is always higher than the write bandwidth. We compute the bandwidth by dividing the size of the transferred data by the measured transfer time in the respective component. Thus, the reason for the different bandwidth is the timing when the matching starts in the reader component relatively to the writer component and is specific to our chosen test cases. As mentioned before, in our experiments the writer always entered the data transfer phase before the reader, hence, it sometimes had to wait for the reader. For this reason, from the writer's point of view, the transfer took longer than from the reader's point of view, hence, the write bandwidth is lower.

The bandwidth using the data transfer does flatten eventually in the strong scaling test (Fig. 8b and c), because the size of the data sent by one process decreases and the overhead of doing the request matching and issuing data requests starts to dominate the transfer time. In the weak scaling tests in Fig. 9 the data transfer time grows slower as the amount of data to transfer by one process stays the same and the overhead of the I/O request matching is relatively small. Hence, the total I/O bandwidth increases faster than in the strong scalability test.



(a) Runtime (b) Write bandwidth (c) Read bandwidth

Fig. 8. Strong scaling of S3D-IO. Y-axis is in logarithmic scale in all plots.



(a) Runtime (b) Write bandwidth (c) Read bandwidth

Fig. 9. Weak scaling of S3D-IO. Y-axis is in logarithmic scale in all plots.

DTF Performance Under I/O Load. Other major factors that impact the data transfer time apart from the number of matching processes are the size of data to transfer and the total number of I/O requests to be matched. To measure the former we perform data transfer for files of various sizes while the number of I/O requests per matcher stays the same. To evaluate the impact of the number of I/O requests, we fix the file size to 256 GB and increase the number of I/O requests a matcher process matches by manipulating how the I/O requests are distributed among matchers. By default, the size of the variable sub-block for which a matcher process matches read and write requests is defined by dividing the variable in the zeroth dimension by the number of matchers. An I/O request is split in the zeroth dimension based on this stripe size and distributed among the matchers in a round robin fashion. In this experiment, we vary the value of the stripe size which effectively changes the number of I/O requests each matcher has to handle.

In both experiments there are 1024 processes per component and there is one matcher per 16 processes. We also note that two out of four variables in S3D-IO have the zero dimension length fixed to 11 and 3, respectively. This is smaller than the number of matchers (64) and results in asymmetrical distribution of work among matchers. For this reason, in the two experiments, on top of the average number of I/O requests per matcher, a small group of matchers has to match approximately 4,000 more I/O requests.

Figure 7 shows the results of the first experiment. The file size was gradually increased from 8 GB to 2 TB. Each matcher process matched on average 576 requests in every test. We measured the time for actual matching of read and write requests—it took only around 2% of the whole data transfer time, thus, we conclude that most of the time was spent on packing and sending the data to reader processes. Thanks to the fast torus network in K computer, sending 2 TB of data over network took less than 3 s.

In the second experiment (Table 1) the file size is fixed, i.e. in every test each process transfers the same amount of data. The matching processes handled from 576 to 16,832 I/O requests, plus the additional requests for some matchers due to the imbalance. We expect that in this experiment it is the request matching process that will have the biggest impact on the data transfer time as the number of requests grow. However, according to the Table 1, the actual request matching took on average no more than 2–3% of the data transfer time and only in the test with 16,832 requests per matcher the matching took around 5% of the data transfer.

Table 1. DTF performance for different number of I/O requests

Average number of I/O requests per matcher	Data transfer time (s)	Time to match read and write requests (s)
576	1.799	0.041
1,088	1.498	0.031
2,144	2.107	0.046
4,224	2.061	0.045
8,448	2.108	0.058
16,832	1.777	0.085

Moreover, we observe that despite the growing number of I/O requests per matcher, the time to perform the data transfer actually decreases in some cases. One explanation for this could be that, when we decrease the stripe size by which the I/O requests are distributed, one matcher becomes in charge of several smaller sub-blocks located at a distance from each other along the zeroth dimension, rather than just one contiguous big sub-block. And this striping may accidentally align better with the I/O pattern of the program, so the matcher ends up matching requests for the data that was written by it. Then, instead of having to forward a data request to another writer process, the matcher immediately can send the data to the reader.

Overall, we conclude that the DTF shows stable performance under increased load of the amount of data that needs to be transferred as well as the load on the matching processes.

4.2 SCALE-LETKF

Finally, we demonstrate how DTF performs with a real-world multi-component application—SCALE-LETKF.

First of all, we explain the I/O pattern of SCALE-LETKF. At the end of one iteration, each ensemble in SCALE outputs the results to two files: a history file and a restart file. At the beginning of its cycle computation, each LETKF ensemble reads the data from the respective history and restart files. LETKF only requires a part of the data in the history file for its computations, i.e. it does not read the whole file. The data transfer function is invoked once per each of the two files. The tests are performed with one iteration because currently SCALE-LETKF does not support the multi-cycle execution.

In the chosen test case LETKF assimilates the data from a Phased Array Weather Radar [11] with a resolution of 500 m. The number of processes participating in one ensemble simulation is fixed to nine processes in all tests, the total number of processes per component is nine multiplied by the number of ensembles. The DTF is configured so that every process in the ensemble acts as a matcher. Additionally, the data buffering is enabled in DTF because the I/O in SCALE happens in several stages and the user buffers are overwritten by the time the data transfer function is called.

The size of the history and restart file in one ensemble is constant, we change the total amount of I/O by varying the number of ensembles from 25 to 100. Table 2 contains the information about the amount of data written and read in each configuration. In all tests, every ensemble process in SCALE writes 363 MB of data, out of which LETKF process requires only about one quarter. A SCALE process generates 255 write requests, LETKF process—31 read request.

Table 2. Cumulative I/O amount in SCALE-LETKF

Number of ensembles	Total write size (GB)	Total read size (GB)
25	79.78	18.97
50	159.56	37.94
75	239.35	56.91
100	319.13	75.88
Per process	363 MB	86.3 MB

Figure 10 shows the execution results of all configurations. Because SCALE-LETKF has a strict timeliness requirement, we focus on the time it took to perform the I/O rather than bandwidth. Additionally, we plot the standard deviation of I/O time between the processes because each ensemble performs I/O independently from each other.

The results show that the DTF helps to improve the total execution time of SCALE-LETKF (Fig. 10a) by cutting on the I/O time. In the largest execution with 100 ensembles, the I/O time was improved by a factor of 3.7 for SCALE and 10 for LETKF. This improvement is rather modest compared to what we observed in tests with S3D-IO mostly because SCALE-LETKF, along with its I/O kernel, is a much more complex application compared to the benchmark.

Apart from the I/O time, we also noticed that, when using the data transfer, the standard deviation decreases significantly compared to when file I/O is

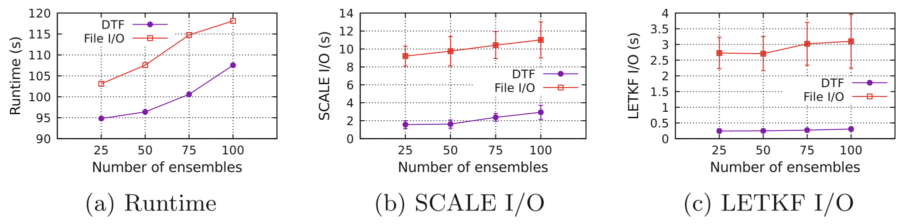


Fig. 10. SCALE-LETKF performance with DTF and file I/O.

used. This has a positive effect on the overall execution, because after LETKF has received all the data it performs global communication over all ensemble processes and smaller deviation in I/O time means that there should be less waiting for processes in other ensembles.

Finally, we note that SCALE-LETKF is still in the stage of development and the most recent version does not meet the target time requirement of 30 s per iteration as can be seen in Fig. 10a. However, we believe that our framework can be of great use to SCALE-LETKF and similar applications and it can help them achieve the execution goal by cutting on I/O time.

5 Related Work

A number of works has addressed the data movement problem, the file I/O bottleneck in particular, in multi-component workflows. Different coupling toolkits have been designed for such workflows [12], especially in Earth sciences [13–15] applications. Such toolkits often provide not only the data movement feature but also allow to perform various data processing during the coupling phase, such as data interpolation or changing the grid size.

For example, DART [16] is a software layer for asynchronous data streaming, it uses dedicated nodes for I/O offloading and asynchronously transferring the data from compute nodes to I/O nodes, visualization software, coupling software, etc. The ADIOS [17] I/O library is built on top of DART and provides additional data processing functionality. However, both, DART and ADIOS require to use a special API for I/O. In additional, ADIOS uses its own non-standard data format for files.

Other coupling approaches include implementing a virtual shared address space accessible by all the components [18], or using dedicated staging nodes to transfer the data from compute job to post-process analysis software during the runtime [19]. In [20], the authors propose a toolkit utilizing the type-based publisher/subscriber paradigm to couple HPC applications with their analytics services. The toolkit uses a somewhat similar concept to inter-component data transferring as proposed in this work, however, they rely on the ADIOS library underneath which the coupling toolkit was built which includes the description of the I/O pattern of the components. Additionally, in our work the matching process is simpler in a way that it takes fewer steps to perform the data transfer.

Providing support to multi-component executions on a system-level is another approach to facilitating the inter-component interaction [21,22]. Current HPC systems usually do not allow overlapping of resources allocated for one executable file. Thus, each component in a multi-component workflow ends up executing on a separate set of nodes and, consequently, the problem of data movement between the components arises. But, for example, in cloud computing, several virtual machines can run on the same node and communicate with each other via shared memory or virtual networking. It has been previously proposed to use virtualization techniques in HPC as well. For example, in [21], the authors show that such virtualization can be used in an HPC environment to allow more efficient execution of multi-component workflows with minimal costs. However, the virtualization is not yet widely adopted in HPC systems.

The main difference of our solution with the I/O library approaches like ADIOS is that such libraries usually provide their own I/O API and underneath that API they can switch between different standard I/O libraries or even perform direct data coupling at user's will. It is assumed that the programmer of the application used this special API during the development stage. In case the application originally used a different I/O library, the I/O kernel must be rewritten. But this may sometimes require a lot of effort, especially when component applications were developed by a third party. Our goal was to provide a simple framework that would allow to switch from file I/O to data transfer with minimal efforts and without having to rewrite the I/O kernels of the workflow components. The DTF operates underneath the PnetCDF library which is a popular I/O library. And while it does not provide as wide functionality as some more advanced coupling libraries, for cases where a user wants to compose a workflow consisting of applications developed relatively independently but all using PnetCDF for I/O, the DTF can work as a quick plug-in solution for faster coupling. The closest solution that we are aware of is the I/O Arbitration Framework (FARB) proposed in [23]. However, the framework was implemented for applications using NetCDF I/O library, that is, it assumes the file-per-process I/O pattern and a process-to-process mapping of data movement. Moreover, during the coupling stage in FARB, contents of the whole file were transferred to the other component's processes regardless of whether the process actually required the whole data or not. In our work, we determine at runtime what data needs to be transferred and only send this data.

6 Conclusion

Multi-component workflows that consist of tasks collaborating with each other to perform computations are becoming a common type of applications running in HPC environments. However, because the current HPC systems are often designed with monolithic applications in mind, it is necessary to determine the main obstacles that prevent multi-component workflows from running at maximum performance in these systems and find solutions. One of the most important issues is the data movement between the components and a number of solutions have been proposed to date.

In this work we proposed one such solution—a data transfer framework called DTF to speed up the data movement between the components in workflows that use PnetCDF API for file I/O. The DTF intercepts the PnetCDF calls and bypasses the file system by sending the data directly to the corresponding processes that require it. It automatically detects what data should be sent to which processes in the other component through a process of I/O request matching.

The DTF requires minimal efforts to start using it in a workflow: There is no need to modify the original PnetCDF calls, rewrite the code using some special API or provide the description of the I/O pattern of the components. The DTF only requires that the user compiles the components using our modified version of the PnetCDF library, provides a simple configuration file listing the files that need to be transferred and adds a few lines to the components' source code in order to enable the data transfer.

Through extensive testing we demonstrated that the DTF shows stable performance under different conditions. However, we believe there is a room for improving the load balancing of the I/O request matching, in particular, the way I/O requests are distributed among the matching processes.

Additionally, due to the fact that the current version of SCALE-LETKF does not support a multi-cycle execution, evaluation of the DTF in such an execution setting is also left for the future work. However, the results we obtained so far are promising and should help SCALE-LETKF to achieve its real-timeliness requirement.

References

1. LANL, NERSC, S.: APEX Workflows. White Paper (2016)
2. Deelman, E., Peterka, T., Altintas, I., Carothers, C.D., van Dam, K.K., Moreland, K., Parashar, M., Ramakrishnan, L., Taufer, M., Vetter, J.: The future of scientific workflows. Int. J. High Perform. Comput. Appl. (2017). https://doi.org/10.1177/1094342017704893
3. Miyoshi, T., Lien, G.Y., Satoh, S., Ushio, T., Bessho, K., Tomita, H., Nishizawa, S., Yoshida, R., Adachi, S.A., Liao, J., Gerofi, B., Ishikawa, Y., Kunii, M., Ruiz, J., Maejima, Y., Otsuka, S., Otsuka, M., Okamoto, K., Seko, H.: Big data assimilation; toward post-petascale severe weather prediction: an overview and progress. Proc. IEEE **104**(11), 2155–2179 (2016)
4. Argonne National Laboratory and Northwestern University: Parallel NetCDF (Software). http://cucis.ece.northwestern.edu/projects/PnetCDF/
5. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.1 (1995). www.mpi-forum.org/docs/
6. UNIDATA: Network Common Data Form. http://www.unidata.ucar.edu/software/netcdf/
7. Mehta, D.P., Sahni, S.: Handbook of Data Structures and Applications. Chapman & Hall/CRC, Boca Raton (2004)
8. Liao, W.k., Choudhary, A.: Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008. IEEE Press, Piscataway (2008)

9. Kurokawa, M.: The K computer: 10 peta-flops supercomputer. In: The 10th International Conference on Optical Internet (COIN 2012) (2012)
10. Ajima, Y., Sumimoto, S., Shimizu, T.: Tofu: a 6D mesh/torus interconnect for exascale computers. Computer **42**(11), 36–40 (2009)
11. Ushio, T., Wu, T., Yoshida, S.: Review of recent progress in lightning and thunderstorm detection techniques in Asia. Atmos. Res. **154**, 89–102 (2015)
12. Dorier, M., Dreher, M., Peterka, T., Wozniak, J.M., Antoniu, G., Raffin, B.: Lessons learned from building in situ coupling frameworks. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization. ACM, New York (2015)
13. Valcke, S., Balaji, V., Craig, A., DeLuca, C., Dunlap, R., Ford, R.W., Jacob, R., Larson, J., O'Kuinghttons, R., Riley, G.D., Vertenstein, M.: Coupling technologies for earth system modelling. Geosci. Model Dev. **5**(6), 1589–1596 (2012)
14. Larson, J., Jacob, R., Ong, E.: The model coupling toolkit: a new Fortran90 toolkit for building multiphysics parallel coupled models. Int. J. Perform. Comput. Appl. **19**(3), 277–292 (2005)
15. Valcke, S.: The OASIS3 coupler: a European climate modeling community software. Geosci. Model Dev. **6**, 373–388 (2013)
16. Docan, C., Parashar, M., Klasky, S.: Enabling high-speed asynchronous data extraction and transfer using DART. Concurr. Comput. Pract. Exp. **22**(9), 1181–1204 (2010)
17. Lofstead, J., Zheng, F., Klasky, S., Schwan, K.: Adaptable, metadata rich IO methods for portable high performance IO. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–10, May 2009
18. Docan, C., Parashar, M., Klasky, S.: Dataspaces: an interaction and coordination framework for coupled simulation workflows. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010. ACM (2010)
19. Vishwanath, V., Hereld, M., Papka, M.E.: Toward simulation-time data analysis and I/O acceleration on leadership-class systems. In: 2011 IEEE Symposium on Large Data Analysis and Visualization, October 2011
20. Dayal, J., Bratcher, D., Eisenhauer, G., Schwan, K., Wolf, M., Zhang, X., Abbasi, H., Klasky, S., Podhorszki, N.: Flexpath: type-based publish, subscribe system for large-scale science analytics. In: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing 2014, pp. 246–255 (2014)
21. Kocoloski, B., Lange, J., Abbasi, H., Bernholdt, D.E., Jones, T.R., Dayal, J., Evans, N., Lang, M., Lofstead, J., Pedretti, K., Bridges, P.G.: System-level support for composition of applications. In: Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2015. ACM, New York (2015)
22. Kocoloski, B., Lange, J.: Xemem: Efficient shared memory for composed applications on multi-OS/R exascale systems. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015. ACM, New York (2015)
23. Liao, J., Gerofi, B., Lien, G.-Y., Nishizawa, S., Miyoshi, T., Tomita, H., Ishikawa, Y.: Toward a general I/O arbitration framework for netCDF based big data processing. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 293–305. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_22