swMPAS-A: Scaling MPAS-A to 39 Million Heterogeneous Cores on the New Generation Sunway Supercomputer

Xiaoyu Hao, Tao Fang, Junshi Chen, Jun Gu, Jiawang Feng, Hong An, and Chun Zhao

Abstract—With the computing power of High-Performance Computing (HPC) systems having stepped into the exascale era, more complex problems can be solved with scientific applications on a large scale. However, due to the significant performance gap between computing nodes and storage subsystems, suboptimal design for the Input/Output (I/O) module will significantly impede the efficiency of scientific applications, especially for the ubiquitous atmosphere applications. Two-phase I/O implemented in N-to-1 mode creates a serious bottleneck that hinders the scalability for the Model for Prediction Across Scales-Atmosphere (MPAS-A) on the new generation Sunway supercomputer. To address the I/O problem, we apply a custom data reorganization method to enable N-to-M I/O mode to exploit the parallel file system's performance and limit the data transfer among MPI ranks to a restricted scope to alleviate communication overhead. Moreover, we have conducted several methods to accelerate the computations, including the redesign for tracer transport, a hybrid buffering scheme, and a three-level parallelization scheme, which allows MPAS-A to use all heterogeneous computing resources efficiently. Experimental results show admirable scalability and efficiency of our I/O method, which achieves speedups of 41× and 58.9× for input and output compared with the raw I/O method on 30,000 MPI ranks. By scaling MPAS-A to 39 million heterogeneous cores, we demonstrate the necessity of a well-constructed I/O module for a real-world atmosphere application. Speed tests show that our optimization methods obtain good results for computations, and MPAS-A achieves a speed of 0.82 Simulated Day per Hour (SDPH) and 0.76 parallel efficiency of strong scaling with 600,000 MPI ranks.

Index Terms—MPAS-A, atmosphere science, heterogeneous core, sunway supercomputer, I/O

1 Introduction

Numerical simulation is the main method to study frontier problems for atmospheric science by using complicated applications (such as Weather Research Forecast (WRF) [1], Model for Prediction Across Scale-Atmosphere (MPAS-A) [2], Community Atmospheric Model (CAM) [3], etc.). In recent decades, atmosphere applications have gradually evolved to non-hydrostatic models (such as MPAS-A) suitable for global simulations at a high spatial resolution to reduce forecast errors. However, non-hydrostatic models increase computational costs and bring more I/O burdens, significantly slowing the simulation efficiency.

In recent years, heterogeneous architectures that involve many-core computing devices [4], [5], [6] have become the foremost provider of computing power in various modern High-Performance Computing (HPC) systems and have shown their ability to run various kinds of climate or weather models [7], [8]. The new generation Sunway supercomputer is the successor of Sunway Taihulight [9], [10], equipped with new heterogeneous many-core processors SW26010pro, of which hardware and software details are shown in Section 3.4. Many works [11], [12], [13], [14] from different scientific fields have been done on it, proving that the new Sunway can provide enough computing power and scale to conduct simulations for global non-hydrostatic atmospheric models with ultra-high resolutions.

However, the suboptimal-designed Input/Output (I/O) scheme makes MPAS-A poorly scalable [15]. The I/O issue of MPAS-A can be regarded as a two-phase (I/O phase and communication phase) I/O problem that is widely studied. For the I/O phase, the HDF5 [16] and NetCDF [17] libraries are widely used for exchanging data with file systems, and their parallel versions are Parallel HDF5 [16] and PNetCDF [18], respectively, built based on MPI-IO [19]. Furthermore, NetCDF can be built with NetCDF4 features implemented based on Parallel HDF5 to enable parallel I/O. ADIOS2 [20], the second generation of ADIOS [21], can use different I/O techniques and file formats, and it even supports changing the transport scheme at run-time. For the communication phase that delivers data to the right MPI processes, Parallel I/O 2 (PIO2) [22] is a popular framework developed for transparent data redistribution and supports multiple formats. Recently, Software for Caching Output and Reads for Parallel I/O (SCORPIO) [23] has been derived from PIO2, and it supports advanced caching and data

- *Xiaoyu Hao, Tao Fang, Junshi Chen, and Hong An are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China. E-mail: {hxy2018, ftao, cjuns}@mail.ustc.edu.cn, han@ustc.edu.cn.*
- *Jun Gu, Jiawang Feng, and Chun Zhao are with the School of Earth and Space Sciences, University of Science and Technology of China, Hefei, Anhui 230026, China. E-mail: {gj99, jw525000}@mail.ustc.edu.cn, chunzhao@ustc.edu.cn.*

Manuscript received 29 January 2022; revised 28 September 2022; accepted 6 October 2022. Date of publication 17 October 2022; date of current version 16 November 2022.
This work was supported in part by the National Natural Science Foundation of China under Grant 62102389 and in part by the National Key Research and Development Program of China under Grant 2017YFB0202002.
(Corresponding Authors: Hong An, Chun Zhao, and Junshi Chen.)
Recommended for acceptance by F. Petrini.
Digital Object Identifier no. 10.1109/TPDS.2022.3215002

rearrangement algorithms while maintaining the same application programming interfaces. It also introduces ADIOS2 as a new backend. However, improper use of these excellent technologies can still expose applications to severe I/O problems, so this work presents what an efficient and scalable I/O module should be to address I/O bottlenecks for a real-world atmosphere application.

Implementing MPAS-A's I/O module with N-to-1 mode causes expensive I/O and communication overhead for simulation at ultra-high horizontal resolution with terabyte-level data. The biggest problem is that N-to-1 mode can not efficiently use the bandwidth of a parallel file system, which is a common problem of parallel file systems on current supercomputers, and the new Sunway supercomputer is not an exception [24], [25]. Communication overhead involves two parts. First is decomposition initialization, where I/O tasks must communicate with all other MPI processes because every MPI process needs to know where its data should be sent to or received from. Correspondingly, decomposition introduces extra computation overhead to establish the mapping relationships of cell data with the I/O tasks. On the other hand, terabyte-level data is communicated among all MPI processes, especially with an irregular topology caused by the cell order in file not strongly associated with the order required by MPI processes.

The practical way to improve I/O performance is to use multiple-file I/O. Weather Research Forecast (WRF) [1] can switch I/O mode between N-to-N (a file per MPI rank) and N-to-M (with quilt servers [26]), and Energy Exascale Earth System Model (E3SM) is built with SCORPIO and ADIOS2 that enables N-to-M I/O mode. Please note that SCORPIO has not yet supported reading multiple files, which can become a potential bottleneck for input because of N-to-1 mode. It also adds extra work for merging checkpoints written as several files into a file if we want to recover the simulation to a specific point. Overall, N-to-M methods can improve I/O bandwidth by adding more files and aggregating I/O requests to make larger I/O operation sizes that contribute to the I/O bandwidth [27]. However, each file can be accessed by only one I/O task in these designs, so increasing file counts (using a larger M) is the only way to promote I/O performance for solving more complex problems with larger I/O volume. The I/O performance increases this way until it reaches a certain point where serving requests to too many files becomes the problem that creates contention for accessing the file system [27] or serialized operations for metadata [28], or both. In addition, every single file being read or written serially by only one MPI task discards the feature of main-streaming I/O libraries that can access a file with several MPI tasks, such as PNetCDF and NetCDF4. In this case, performance can be slow for large file sizes when creating more files is not allowed, but it is not absolute, as I/O performance depends on I/O libraries and the underlying parallel file system that is transparent for most users. Thus, we design the I/O module for MPAS-A in a more flexible way that supports finding the optimal performance by tuning the number of files and MPI processes that access each file. It also supports reading and writing multiple files without breaking the original simulation workflow.

Besides improving I/O performance, we have conducted some methods custom to the new Sunway with heterogeneous architecture to further accelerate the computations for MPAS-A. The most straightforward way of reducing simulation time is to increase the scale. Since there are a certain number of cells around the globe, scaling up the system reduces workloads on individual computing units, allowing for shorter running time. However, most computing procedures of MPAS-A, particularly in the dynamic core, have low arithmetic intensity and suffer the problem of non-continuous or random memory access that destroys the data locality. Tracer transport, the most time-consuming procedure in the dynamic core, communicates a small amount of data but frequently in halo exchanges, which is not conducive to bandwidth utilization. Furthermore, how to fully utilize the computing power of heterogeneous cores is crucial. We take a lot of efforts to overcome the above challenges. The main contributions of this paper are summarized as follows:

1) We break down the I/O time of MPAS-A during the large-scale execution and show the factors that result in the I/O bottleneck, which provides suggestions for improving the I/O performance of other applications.

2) We propose a custom data reorganization method that helps to convert I/O mode from N-to-1 to N-to-M and alleviates communication overhead to address the two-phase I/O problem of MPAS-A. We demonstrate that a well-designed I/O scheme can significantly contribute to file system usage by scaling MPAS-A to 600,000 MPI ranks on the new generation Sunway supercomputer. Experimental results illustrate the efficiency and scalability of our I/O scheme.

3) We apply methods to accelerate the computations for MPAS-A including tracer transport redesign and hybrid buffering scheme. Moreover, a three-level parallelization scheme is performed to utilize the heterogenous cores' computing power, enabling MPAS-A to run on 39 million cores on a scale of 600,000 MPI ranks.

4) We report the speed and parallel efficiency of MPAS-A on a extreme large scale of 600,000 MPI ranks.

The remainder of this paper is organized as follows. Section 2 presents the related works of atmospheric simulations at ultra-high resolution with corresponding optimizations and challenges. Section 3 briefly introduces the MPAS-A, its I/O module, and the new generation Sunway supercomputer. In Section 4, our I/O scheme and acceleration methods are provided. Sections 5 and 6 present experimental results and the conclusion, respectively.

2 Related Work

2.1 I/O Bottleneck for Atmospheric Application

I/O bottleneck is a common issue for atmospheric simulation applications that need to read or write non-contiguous blocks in multiple array variables, particularly for ultra-high resolution. D. Heinzeller et al. [15] have conducted an extreme scaling test on Juqueen, which successfully scales MPAS-Atmosphere (MPAS-A) to 28672 nodes of 458,752 MPI tasks and achieves 69.5% parallel efficiency of strong scaling for U3KM with I/O. They report that the overall model initialization before time integration takes 48 minutes, and 10-15% of time is approximately spent on initialization and I/O. Their work has shown that scaling MPAS-A to a vast system is possible with initialization and

I/O stages. Another work [29] scales MPAS to 4,096 Xeon Phi processor cores on Nurion supercomputer. They report that the MPAS has excellent scalability for time integration, but they also face a severe I/O bottleneck issue when trying two additional I/O strategies (i.e., adjusting the stripe count and using a burst buffer). Some works start from the two-stage I/O problem, trying to solve the I/O bottleneck [30], [31]. For example, [31] reduces the communication overhead by aggregating non-contiguous requests for array blocks, provides corresponding parameter assignment strategies, and implements their work into E3SM [32], achieving promising results. Yashiro et al. [33] scale NICAM-LETKF to 131,072 nodes on Fugaku. One of their contributions is the optimization of file I/O. They adapt a throughput-aware application design to prioritize data locality. This design maximizes file I/O throughput and divides global communications into smaller ones.

2.2 Acceleration for Atmospheric Application With Heterogeneous Architecture

More and more leadership supercomputers are designed using the heterogeneous many-core-based architecture. Many works focus on accelerating atmospheric simulations on these supercomputers such as Sunway Taihulight, the world's 6th-ranked supercomputer [34], that has similar architecture to the new generation Sunway supercomputer. Fu et al. [35] first refactor and optimize CAM on the Sunway TaihuLight, then scale CAM-SE to the entire system [36] that simulates at a 750m global resolution. Compared with their work that only contains the non-hydrostatic dynamic core, MPAS-A has more physics and chemistry parameterizations, which brings more I/O burdens. Xu et al. [8] port the whole WRF onto Sunway Taihulight on a large scale. They optimize the huge codes of WRF's dynamic core using a domain-specific compiler and use OpenACC directives to accelerate physical parameterizations. Both techniques reduce the human efforts for porting and optimizing programs and achieve high performance. Yang et al. [7] develope a fully implicit solver and successfully scale it to the system of Sunway Taihulight of 10.5M cores at the 488-m horizontal resolution. They further optimize the program using register communication, on-the-fly array transposition, and xMath library. Their work have won the 2016 Gordon Bell Prize. These works have achieved high performance on heterogeneous architecture by employing various optimization schemes, but they did not report the optimization for I/O.

3 Background

3.1 The Global Non-Hydrostatic Atmospheric Model

The Model for Prediction Across Scales-Atmosphere (MPAS-A) [2] is a non-hydrostatic atmospheric model suitable for global simulations at a few kilometres. MPAS-A is developed as a global complement to the WRF. The MPAS-A non-hydrostatic dynamical core discretizes the computational domain horizontally on a C-cell staggered unstructured Voronoi mesh using finite-volume formation [2]. The fully compressible non-hydrostatic equations are cast in terms of geometric-height hybrid terrain-following coordinate, and the solver applies the split-explicit time integration scheme. The time-integration scheme employs a 3rd-order Runge-Kutta method and an explicit time-splitting technique [37]. The MPAS dynamic core has recently been widely used in different atmospheric models to address many important scientific questions [38], [39]. The physics parameterizations of MPAS-A are the same as the WRF model [1]. Nowadays, new chemistry parameterizations are introduced [40] and coupled with the existing physics parameterizations and dynamic core, which adds more computational and I/O burdens that can be a vital issue for modern or future HPC systems. More atmosphere details can be found in [40].

3.2 MPAS-A Workflow

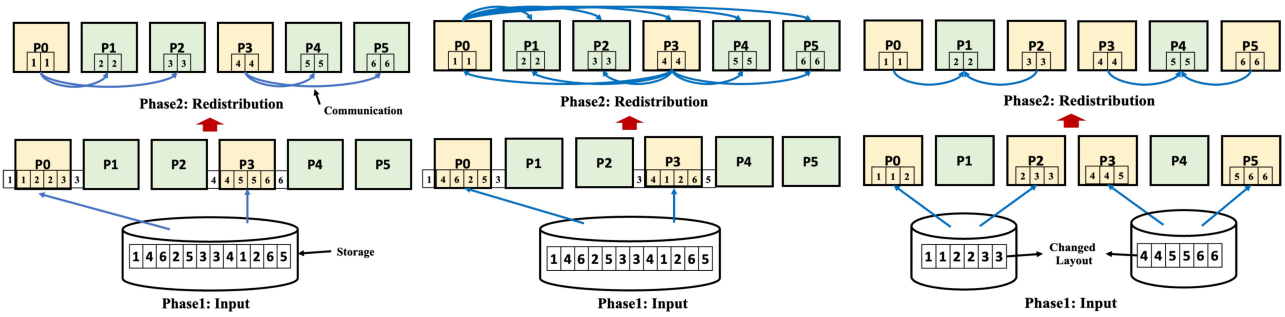
The entire workflow of MPAS-A is shown in Fig. 2a. MPAS-A has a special relationship between input and output. The input file of MPAS-A is generated from the *init_static.nc* that uses the program *init_atmosphere*, sharing the same I/O framework with the actual simulation program *atmosphere*. After simulation, analysis programs are used to get atmospheric results. MPAS-A requires reading initial condition files at the beginning and writing regular outputs and checkpoints at a fixed time interval. Higher resolution means a huge increase in horizontal cell number, significantly increasing I/O volume. As shown in Table 1, V4KM contains 785,410 cells, but U3KM contains 65,536,002 cells. Simulations at the resolution of the U3KM require reading and writing terabyte-level data (3.3T input, 1.1T regular output, and 5.7T checkpoints with chemistry procedures [40]).

3.3 MPAS-A I/O Module With PIO2

I/O Module of MPAS-A is developed based on PIO2 to solve the problem that distributes a global array across several computational units that operate on their own and local subset of the global variable array [22], but it only supports N-to-1 I/O mode. Based on PIO2, MPAS-A supports different file formats such as NetCDF, NetCDF4, and PNetCDF, requiring users to select part of MPI processes as I/O tasks for I/O.

I/O task. I/O tasks aggregate I/O requests from other MPI processes and exchange data with the file systems. The number of I/O tasks is specified by setting the I/O stride. For example, if we select an I/O stride of 30 for 3000 MPI processes, there will be 100 I/O tasks.

Rearrangers. Rearranger controls the way that I/O tasks use to exchange data with the file system, which further affects the data arrangement method between computing tasks and IO tasks. Two rearrangers can be selected, including subset rearranger and box rearranger. Subset rearranger is shown in Fig. 1a. Each I/O task handles I/O requests from its own and neighbouring MPI processes, requiring continuous access to the file system until all data has been written or read. For example (in Fig. 1a), P0 only reads data required by P0, P1 and P2 labelled with 1, 2 and 3, so it requires accessing file totally 4 times. The subset rearranger faces potential I/O bottleneck because of contention of file system resources. Box rearranger is shown in Fig. 1b. All I/O tasks cooperate to handle I/O requests from all MPI processes, avoiding accessing the file system too frequently but requiring massive communications. For example, P0 reads the first half of all the data, and P3 reads the second half before sending it to other MPI processes. Box rearranger is a



(a) Subset (b) Box (c) Ours

Fig. 1. (a) and (b) are two alternative I/O methods used by MPAS-A that are called subset and box. (c) is our application-level I/O scheme built with PIO2, providing an efficient, flexible and scalable way to solve the I/O problem.

better choice for heavy I/O burden and is the default method MPAS-A uses.

Decomposition initialization. Decomposition initialization is based on rearrangers determining how the data will be transferred among MPI tasks through MPI communications. As phase 2 (redistribution) shown in Fig. 1a, two things are decided when initializing decomposition. The first is that P0 should communicate with P1 and P2, and the second is that P0 should keep cell data labelled with 1 but send 2 and 3 to P1 and P2, respectively. Please note that this process can be time-consuming, especially running with box rearranger on large scales and large variable sizes in a file.

3.4 The New Sunway Supercomputer

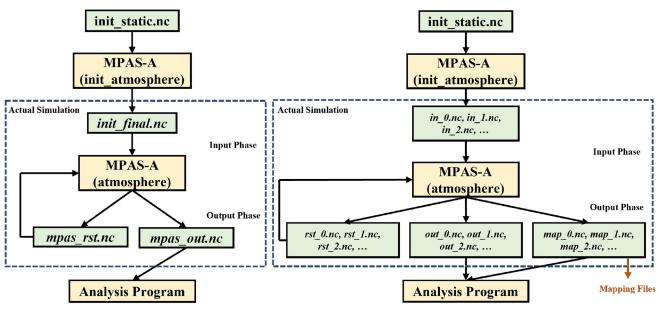
The new Sunway supercomputer inherits and develops the architecture of Sunway TaihuLight [9]. We will discuss its heterogeneous processor, file system, interconnection, and programming environment.

3.4.1 SW26010pro Processor

SW26010pro many-core heterogeneous processor has 390 cores grouped into six core groups (CGs) with 65 cores per CG. Each CG integrates one management processing element (MPE) and one cluster of 8x8 computing processing elements (CPE). CGs of one processor are interconnected through a network on chip (NoC). MPE that operates at a frequency of 2.1 GHz with lower computing power is responsible for handling management and communication functions, consisting of 32KB L1 instruction cache, 32KB L1 data cache, and 512KB L2 cache. CPE is designed to do the computation, which supports double-precision float, single-precision float, half-precision float, and integer arithmetic. Each CPE operates at a clock frequency of 2.25 GHz and has a 32KB instruction cache and a 256KB scratchpad memory (which is also called local data memory (LDM)) that can be partly configured as hardware data cache. CPEs have two execution pipelines, one of which is responsible for scalar/vector operations of float/integer types, whereas another supports only scalar integer computing as well as load/store, jump and compare operations [11]. Moreover, CPE and MPE provide 512-bit and 256-bit SIMD support, respectively. The theoretical peak performance of a core can be calculated following the formula [41]:

$$flops = frequency * 2 * vector\ elements * pipelines$$

, where $vector\ elements$ will be 1 if only scalar operations are performed, and in this case, a CPE is roughly 0.54× the peak performance of an MPE without SIMD but 1.07× of an MPE with SIMD.



(a) Original (b) Refactored

Fig. 2. (a) and (b) present the workflows of MPAS-A before and after refactoring I/O module, respectively. The workflows are the same except for the additional output of mapping files.

3.4.2 File System

As shown in Fig. 3, the file system uses a tiered storage architecture with computing nodes at the top and a parallel file system at the bottom. A Lightweight File System (LWFS) [25] consists of many I/O nodes (ION) in the middle is responsible for performing I/O forwarding service. The role of the I/O node is to aggregate and forward requests from computing nodes to the underlying parallel file system, and approximate 170 compute nodes statically share one I/O node with a max bandwidth of 4GB/s. For a single core group, the I/O bandwidth is about 100MB/s, but for six core groups on a node whose aggregated I/O bandwidth is only 400MB/s. So the peak I/O bandwidth can be roughly computed for a specific scale of N as:

TABLE 1
Configurations for Different Problem Sizes

Problem size	Horizontal resolution	Mesh cells	Vertical	Time step
U60KM	60 km	163842	56	300s
V4KM	4-60km variable	785410	56	20s
U3KM	3 km	65536002	56	15s

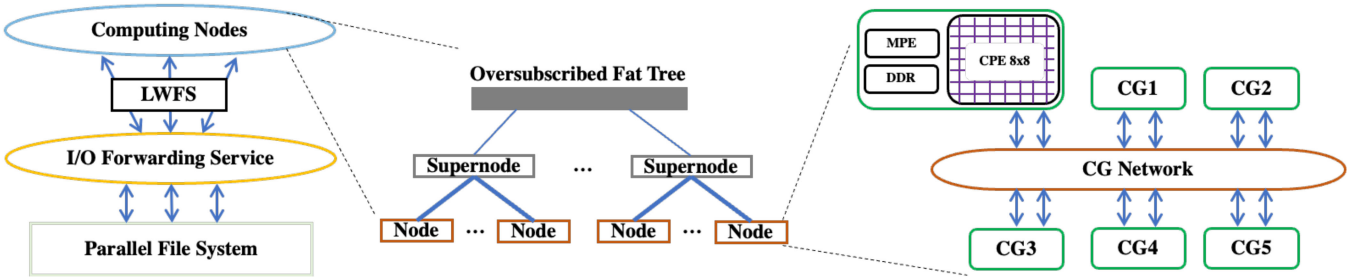


Fig. 3. The architecture of the new generation Sunway supercomputer.

$$bandwidth = N/1024 * 4 + min((N\%1024)/6 * 0.4, 4)$$

For a large N, the effect of the second part is negligible.

3.4.3 Interconnection

The topology for the interconnection is a 16/3x oversubscribed fat tree with a non-blocking fabric. Each supernode contains 256 compute nodes (each node has six core groups) with peer-to-peer connections, and the inter-supernode communication bandwidth is 3/16x of inter-supernode bandwidth [42]. Moreover, the MPI version is customized and optimized for the SW26010pro's heterogeneous architecture.

3.4.4 The Programming Environment

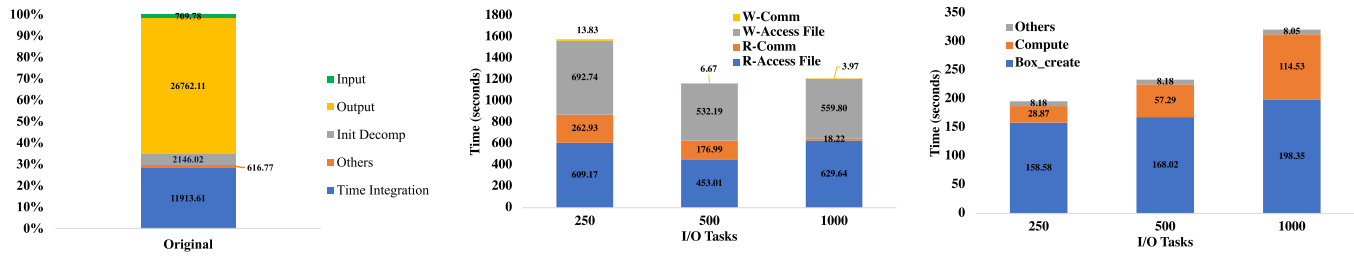
The programming model on the new Sunway supercomputer is a combination of MPI and Athread/OpenACC. Generally, one CG corresponds to one MPI process, and thread-level parallelism can be realized through Athread or OpenACC. Athread is a lightweight threading library designed for Sunway many-core processors. Athread provides manual but fine-grained ways to parallelize computation in both thread-level and data-level (vectorization with SIMD units), manage data transfer between LDM and main memory, and control communications between CPEs through Remote Memory Access (RMA).

4 Methods

4.1 Performance Analysis for MPAS-A

Fig. 4a shows the time percentage of different phases for a one-hour simulation with 30,000 MPI processes and 1000 I/O tasks using MPEs only, which reads data of 1.1 TB and writes data of 6.8 TB. Unsurprisingly, decomposition, reading and writing take most of the time, all of which are related to the I/O. It is unacceptable that I/O occupies more than 70% of the entire wall-time, which is the biggest obstacle for scaling MPAS-A to a large scale. So, to solve the I/O problem of MPAS-A, we present the time breakdowns for the I/O stages and decomposition and provide an in-depth analysis of the causes of the I/O problem. We have performed several runs for the detailed time breakdown, but N-to-1 I/O mode is pretty slow, so the I/O tests are conducted on a smaller scale of 10,000 MPI ranks, decreasing output from 6.8TB to 509GB but maintaining input the same (1.1TB).

I/O time breakdown. Fig. 4b shows the time percentage of accessing the file system and communicating when MPAS-A reads and writes files using 250, 500, and 1000 I/O tasks, respectively. We observe that the percentage of communication time in the read stage is 30% for 250 I/O tasks, 28% for 500 I/O tasks, and only 2% for 1000 I/O tasks, which illustrates a negative correlation trend. In MPAS-A, all MPI ranks get data through communication. With fewer I/O tasks, the fewer senders in reading and receivers in writing, and each I/O task needs to communicate with more other MPI ranks, resulting in more communication time. Moreover, the communication between I/O tasks and compute tasks is irregular due to the workload-balance method with Metis [43], even in the fully connected manner, as shown in Fig. 1b. The communication time for writing has the same trend but is not as time-consuming as reading because it writes less data than reading in these tests. Increasing the number of I/O tasks reduces the communication time and exploits more I/O bandwidth until it meets the bottleneck created by the file system, just as the performance with 1000 I/O tasks shown in Fig. 4b. Thus, optimal I/O performance requires carefully selecting the number of I/O tasks. Optimizing the communication time is necessary when assuming that running with 500 I/O processes is the optimal choice.



(a) Execution Time Breakdown (b) I/O Time Breakdown (c) Decomposition Time Breakdown

Fig. 4. (a) is the time breakdown of running MPAS-A for one hour of simulated time. The I/O costs more than 70% of the entire wall time. (b) shows the time percentage of accessing the file system and communication for the reading and writing stages. **R-** and **W-** prefixes represent reading and writing, respectively. In (c), **Box_create** is a procedure of communication, and **Compute** is the computational part in decomposition that builds the relationship between cell data and I/O tasks.

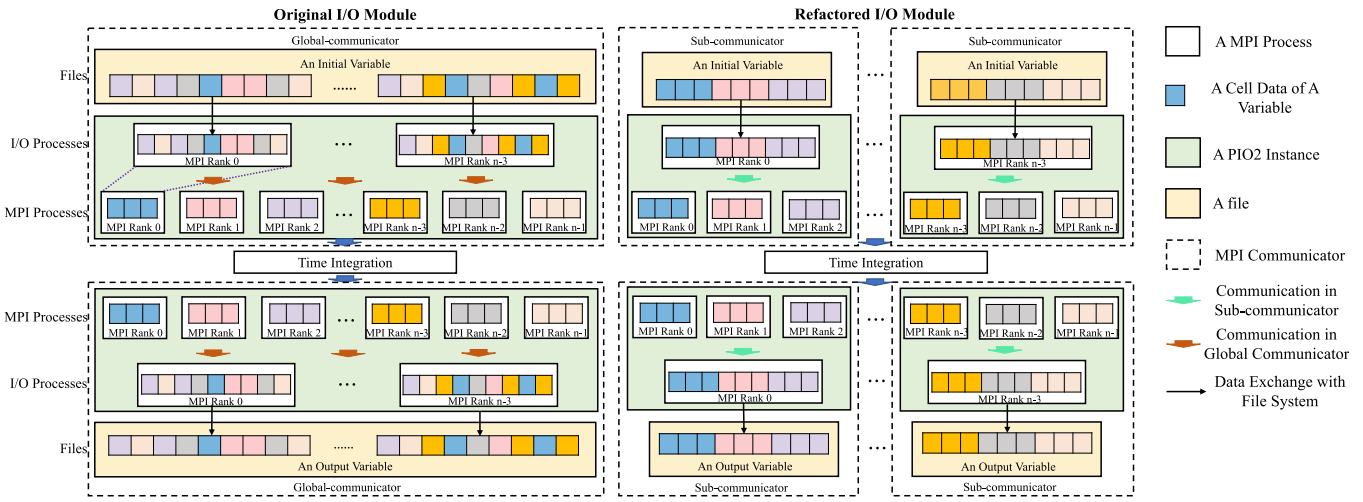


Fig. 5. Comparison of the original and refactored I/O module of MPAS-A. The left panel shows the original I/O module of MPAS-A, in which communication occurs in a global communicator and involves all MPI processes. The right panel is our refactored I/O module in which communication only occurs in each sub-communicator, which is achieved through our novel partition method that puts data required by MPI processes in a sub-communicator into one file.

Decomposition time breakdown. Decomposition takes more time than reading, so it is essential and helpful to figure out why decomposition takes so much time. Fig. 4c shows the time breakdown of decomposition for the different number of I/O tasks, where communication is the most time-consuming part, followed by the computation that builds and stores the mapping between the cell data and I/O tasks. Communication expenses are high because each MPI rank needs to know where its data comes from (in reading) and send to (in writing), which relies on the I/O tasks communicating with all other MPI processes. In this case, the total number of communications is $M * N$, where M is the number of I/O tasks and N is the MPI ranks' number, so the communication time in Fig. 4b increases with more I/O tasks. Moreover, the time complexity of the computation is about $\mathcal{O}(M * D)$, where D is the count of cells processed in each MPI process. Fixing the number of MPI ranks, D will be constant, and increasing the number of I/O tasks will lead to an increase in time, as shown in Fig. 4c.

4.2 Refactoring I/O Module

This subsection presents how the I/O module is refactored based on a reorganized data layout that helps easily enable I/O with multiple files. From the view of contents in files, multiple files used by the refactored I/O module can be regarded as splitting the original large single file used by the original I/O module into many separate files, but in practice, the partition procedure is not independent of MPAS-A workflow as shown in Fig. 2b.

4.2.1 Data Layout

Contents in each file for multiple-file I/O should be managed carefully to address the problems discussed in Section 4.1. The new data layout in the file is established based on the original static workload balance method in MPAS-A with Metis. If we combine the contents of all multiple files used by refactored I/O module in order, then each MPI process can read (write) all data from (to) the file at once according to the offset (where the required cells start) and count. The idea for reorganization is simple: store cell data according to the MPI rank instead of the order of cells in the globe. We have achieved it by putting together the data needed by the same MPI processes in a file. After reorganization, we can "partition" the original single file into multiple files by aggregating data from the arbitrary scope of adjacent MPI processes and then store them following the order of MPI rank. The data layouts before and after reorganization refer to the storage part in Figs. 1a and 1c, respectively. I/O with multiple files takes two advantages of the new layout. First, data required by an MPI process is stored in one file to avoid accessing too many files. Second, communications only exist among MPI processes whose data is regularly stored in the same file. It limits communication to a restricted number of MPI processes depending on the cell number in each file. Moreover, it takes advantage of the intra-supernode bandwidth that is higher than the inter-supernode bandwidth on the new Sunway supercomputer. Given that the total I/O volume is fixed, the more files there are, the smaller the individual file size, thus reducing the amount of communication in each MPI processes scope.

4.2.2 Implementation

The implementation is based on PIO2, which enables accessing one file with more than one I/O task based on multiple-file I/O with less concern of metadata bottleneck. Please note that the number of I/O tasks per file is configurable and can be tuned by users to achieve the best I/O performance.

One should put all MPI processes into one MPI communicator to read or write one file in PIO2, which is how the original MPAS-A I/O module is realized, as shown in the left panel in Fig. 5. To implement multiple-file I/O, we first create multiple sub-communicators whose count is the same as the number of files, and we initialize a PIO2 instance in each sub-communicator to manage a single file. In order to minimize the impact on other functional parts of MPAS-A, we reserve the global communicator that includes

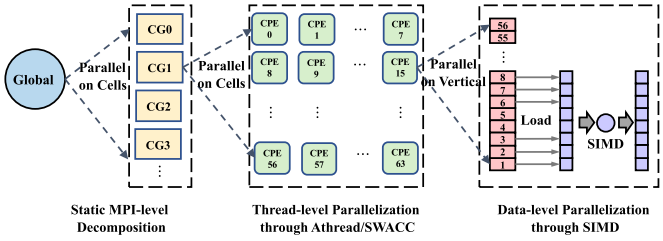


Fig. 6. Three-level parallelization scheme for MPAS-A. The first two levels decompose horizontal cells to different CGs and CPEs, respectively, but the second level achieves data parallelization on vertical layers through SIMD.

all MPI processes. The refactored I/O module turns the I/O pattern into $N - M * S - M$, where N is the number of MPI ranks, M is the number of files, and S is the number of I/O tasks in each sub-communicator that controls the number of I/O tasks that indeed access the file system. $M * S$ is the total number of I/O tasks, which is less than or equal to N. We present our refactored I/O module in the right panel of Fig. 5.

There are a few details that are worth being mentioned. In both the input and output phases, we must perform a decomposition initialization before accessing the file system, which requires that all MPI processes in each sub-communicator know the offsets (or positions) of the data being processed in that file by calling *MPI_Allreduce*. Moreover, when reading the input file, we should call *MPI_Allreduce* to accumulate the global dimensions of some variables that describe the geometric information for the mesh because these values are related to the size of the variable array stored in files. Furthermore, to ensure offsets are correct, we need to keep the scale and file number of *atmosphere* the same as that of *init_atmosphere*.

Unfortunately, the refactored I/O module stores variable arrays according to MPI rank, but the cells in MPI tasks are not in the same order as in the globe, which breaks the positional relationships between the cells. Therefore, to make it possible to analyze the atmospheric results, we additionally generate many *mapping files* that build the mapping relationships between local indexes in new small files and global indexes in the original globe, as shown in Fig. 2b. Those mapping files work for all input or output files, and we only need to generate them once for a specific scale and file number, no matter how many files are written.

4.3 General Parallelization Scheme

Original physics and newly introduced chemistry parameterizations [40] in MPAS-A are the same as WRF [1] and WRF-chem [44] respectively, but they were modified to be consistent with the code style of the dynamic core. Therefore, these three computation parts can take a similar parallel approach. Nevertheless, compared with dynamic core, physics and chemistry parameterizations have more local temporary variables. Consequently, the total space of temporary variables in some computing procedures exceeds the capacity of CPE's local storage, or even a single variable cannot be entirely allocated. Thus we allocated these kinds of local variables in the main memory, which requires an extra data swap between LDM and the main memory, and we use hardware cache to do it. The parallel scheme is shown in Fig. 6 and detailed as follows:

(1) MPAS-A, by default, uses the static load balancing method by reading a graph partition file generated by an external tool Metis, which chunks global cells into many blocks and assigns each of these blocks to one MPI process. Thus, the domain decomposition at the MPI level is no longer required to be done manually. At a specific scale, cells assigned to each MPI process are fixed, and corresponding vertical layers of a cell are put entirely into one MPI process without partition. Each CG is responsible for one MPI process.

(2) Compared with the number of cells, the number of vertical layers is relatively small as shown in Table 1. For example, the average number of cells in each MPI process is 2184 in horizontal and 56 layers in vertical at a scale of 30,000 MPI processes for U3KM. The vertical layer's number does not change according to the scale, and it is the axis where the arrays are stored along. Please note that the vertical layer is tunable according to different research purposes. Therefore, we only tile the cells assigned to each CG along the horizontal dimension to realize thread-level parallelism. Each CPE in a CG runs asynchronously to compute 1/64 of all workloads assigned to this MPI process. The asynchronous implementation makes a CPE load its data once it is ready to conduct computation, avoiding the extra cost of potential synchronizations like one CPE waiting for others to finish.

Listing 1: Compute divergence at each cell center. Manual data transfers are implemented through *copyin* and *copyout*, and variables not mentioned use cache directly, which is the key point of our hybrid buffering scheme. We use *tile* to automatically tile loops

```
!$ACC parallel loop copyout(ke)
!$ACC tile(4)
!$ACC copyin(edgesOnCell_sign, nEdgesOnCell)
!$ACC copyin(edgesOnCell)
!Variables not in copyin will use cache
do iCell=cellStart,cellEnd
    ke(:,icell) = 0
  do i=1,nEdgesOnCell(iCell)
    !iEdge is non-continuous
    iEdge = edgesOnCell(i,iCell)
    s = edgesOnCell_sign(i,iCell) * dvEdge(iEdge)
    do k=1,nVertLevels
      divergence(k,iCell) = \
        divergence(k,iCell) + s * u(k,iEdge)
      ke(k,iCell) = \
        ke(k,iCell) + 0.25 * ke_edge(k,iEdge)
    end do
  end do
!Following computing parts are omitted
end do
```

(3) After completing the thread-level parallelization, we try to conduct data-level parallelization for computational kernels in the dynamic core by manually vectorizing the loop of the vertical layer, because the computational and memory-access patterns among different computing procedures in the dynamic core are roughly the same, with the most inner loop iterating over the vertical layer. However, manual vectorization takes much effort but yields only a

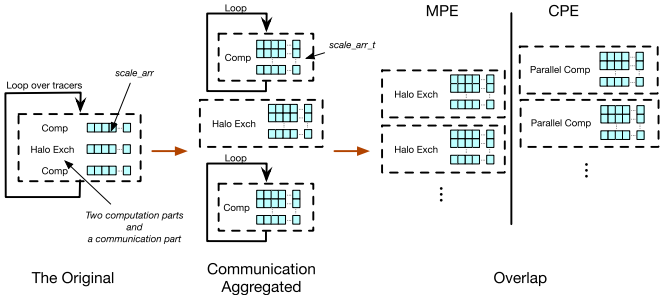


Fig. 7. Tracer Transport Redesign. The leftmost panel is the original tracer transport algorithm, the middle panel is the algorithm after aggregating communication, and the rightmost panel is the redesigned tracer transport that overlaps communication and computation.

tiny margin of performance improvement or even produces negative optimization, possibly due to the computations of low arithmetic intensity in the dynamic core. CPE supports 512-bit SIMD, which requires loading extended vector-type variables (double vector, int vector, etc.) with values of corresponding standard-type variables (double, int, etc.). These variables must be allocated or loaded in LDM (hardware data cache is not allowed for SIMD), wherein vectorization involves a memory copy. However, the computations of MPAS are memory-intensive: most of the operations are element-wise, with the data involved discarded after being used once, which results in a low data reuse rate, and most of the time is spent on data transfer or copy, preventing the full use of SIMD.

4.4 Tracer Transport Redesign

Tracer transport is the most time-consuming procedure in the dynamic core, which spends most of its time on halo exchanges. As shown in the leftmost panel of Fig. 7, the tracer transport iterates over tracers, and in each iteration, the halo exchange communicates an array named *scale_arr* within two computation parts. Here, *scale_arr* is a 3D variable. We start with parallelizing the two computation parts on CPEs directly, but it is inefficient because it launches the CPE kernel as the same time as the loop number, causing non-negligible overhead. Moreover, this kind of communication that delivers a little volume of data but occurs frequently is not conducive to bandwidth utilization.

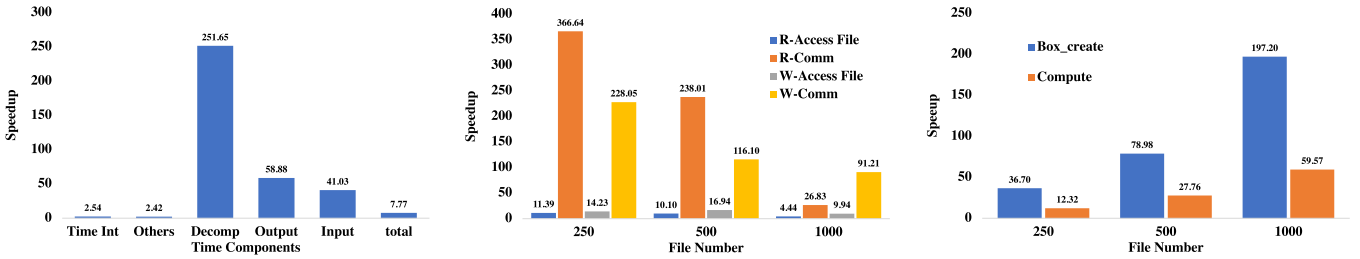
To address the above problems, we redesign the tracer transport algorithm by aggregating multiple halo exchanges from different loop steps and then overlapping them with the first computation part. Some temporal variables such as *scale_arr* are reused in different iterations, leading to the lack of parallelism between tracers. In order to eliminate the dependencies between iterations, we introduce extra buffers for those temporal variables to store the results of each iteration. In this approach, halo exchange is decoupled with computation parts, and arrays of *scale_arr* computed from different tracers are aggregated into a large *scale_arr_t* (in the middle panel of Fig. 7). Now we can reduce the number of halo exchanges and increase the volume of data in a halo exchange. We then overlap the first computation part with halo exchange by tilling the loop of tracers and aggregating *scale_arr* of each tile. The overlap is implemented by inserting a halo exchange between *athread_spwan* that launches CPE kernels and *athread_join* that synchronously waits for all CPEs finishing their works. The second computation requires waiting for the halo exchange to finish, so we let it run alone. Note that we swap the loops of cell and tracer when parallelizing the computation on CPE, which is more efficient because the number of cells is significantly larger than tracers.

4.5 Hybrid Buffering Scheme

Compared with the previous generation, the local storage in each CPE of SW26010pro is enlarged to 256KB and can be partly configured as a hardware data cache (0KB, 32KB, and 128KB), which delivers effort-free performance improvement. However, most computing kernels in the dynamic core have a common issue: non-continuous memory access corrupts the cache's data locality. For example, codes in Listing 1 compute the divergence at each cell center, which needs read arrays *u* and *ke_edge* that are indexed by edges as shown in Lines 14 and 16 respectively. However, *iEdge* is obtained discontinuously, as shown in Line 10. To address this issue, we propose a hybrid buffering scheme to optimize this kind of loop feature. The scheme is simple but efficient that puts arrays used sequentially in LDM and leaves other non-continuous-access arrays to cache. Moreover, we tile the loops into blocks to efficiently use memory bandwidth. The scheme is implemented depending on the programming syntax of Athread or OpenACC. Most computing kernels in the dynamic core are realized through OpenACC, but the redesigned tracer transport is implemented through Athread. As shown in Listing 1, we use compilation guidance statements supplied by OpenACC to implement a hybrid buffering scheme, where *copyin* and *copyout* are used for loading read-only arrays into LDM and put write-only arrays from LDM to memory respectively. Arrays of name not mentioned in statements use cache directly like *ke_edge*, *dvEdge*, and *u*. OpenACC also provides a convenient approach to tile the loop automatically through *tile* statement. Some kernels require reading or writing a total of a dozen arrays. Consequently, we are facing the choice of reducing the block size to load more arrays into LDM or keeping the block so large that only part of the arrays can be loaded. According to extra tests, we choose the latter one, which loads read-only and write-only arrays into LDM, because compared with the arrays involving both reading and writing, it causes less data swap between LDM and main memory. The codes shown in Listing 1 are computed with several arrays such as *ke*, *divergence*, etc. Providing that we can put part of variables into LDM with a tile size of 4 but all of them with a tile size of 2, then we prefer the former and load LDM with read-only and write-only arrays like *ke*, *nEdgesOnCell*, and *edgesOnCell_sign*. *divergence* needs to move from main memory to LDM and back to main memory after modification. *nEdgesOnCell* and *edgesOnCell_sign* are read-only, so they only need to move from main memory to LDM once, and *ke* can be directly allocated in LDM and then written back to memory after modification.

5 Experiment Results

We evaluate the performance of the MPAS-A on the new generation Sunway supercomputer, starting with the



(a) Speedup for Execution Time (b) Speedup for I/O (c) Speedup for Decomposition

Fig. 8. Speedups for different time-consuming parts discussed in Section 4.1.

overall performance comparison between original and optimized codes. The speedup is shown in Fig. 8a. The time components are related to what shown in Fig. 4a. The total time is improved by 7.77× with time integration achieving 2.54× speedup. For I/O, decomposition initialization, input and output achieve speedups of 251.7×, 41×, and 58.9×, respectively.

5.1 I/O Performance

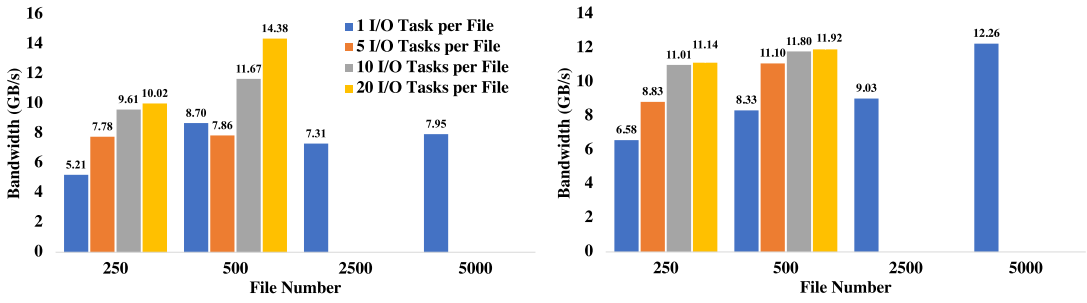
5.1.1 Speedup

Reading and writing stages. The detailed improvements of I/O are presented in Fig. 8b. The communications in reading and writing are substantially improved. Our I/O scheme limits the communication to a small scope that takes advantage of the higher bandwidth within the supernode and less communication volume. The performance of accessing the file system has also been improved. It is worth mentioning that the aggregated I/O bandwidth is affected by the I/O node statically assigned to the compute node. With more MPI ranks, more I/O nodes will be allocated, and higher I/O performance can be achieved. We also show the impact of writing files with the different number of I/O tasks. Two data formats that support parallel I/O are chosen, including NetCDF4 and PNetCDF, and their results are presented in Figs. 9a and 9b, respectively. Test for data formats that do not support parallel I/O is unnecessary, such as Serial NetCDF. According to Fig. 9a, NetCDF4 achieves the best write performance with 500 files and 20 I/O tasks per file, which applies all MPI processes to do I/O. Interestingly, this configuration uses more I/O tasks than 5,000 files with 1 I/O task per file but achieves better I/O bandwidth. Results for PNetCDF are slightly different. Using more files brings higher I/O bandwidth until it creates bottlenecks for the file system, but I/O with fewer files can achieve better performance by tuning the number of I/O tasks per file, and NetCDF4 has more significant performance gains from it.

Decomposition Initialization. The speedup for the decomposition is shown in Fig. 8c corresponding to Fig. 4c. The speedup is significant because enabling multiple-file I/O restricts the MPI ranks' number involved in decomposition, reducing both the communication and computation overhead. Improvement for communication is better than computation because refactoring I/O only affects the number of I/O tasks of a file and does not change the number of cells required by MPI processes. Moreover, Table 2 shows the acceleration results we achieved to decompose the variable *qv* at larger scales. In this experiment, the largest chosen scale is 120,000 MPI processes to finish recording initialization decomposition in an acceptable time. In the original I/O module, the decomposition time of variable *qv* increases when enlarging the scale, up to 576.6s at 120,000 processes. We can imagine how colossal time will be spent to decompose all dozens of variables. The refactored I/O module significantly decreases the decomposition time and even makes a negative correlation trend when enlarging the scale. The number of MPI ranks that is the same as the size of sub-communicator and I/O task that is 1 involved in decomposition is fixed, so the communication overhead is roughly the same, but running with more MPI ranks decreases the cell number in each MPI rank, which reduces the computation overhead and then achieves a better performance.

5.1.2 I/O Scaling Performance

To explore the scalability of our refactored I/O module, we test the I/O performance at different scales for input and output bandwidth. In our experiments, we first use the



(a) NetCDF4 (b) PNetCDF,CDF5

Fig. 9. I/O bandwidth tests for different file numbers with different I/O task numbers. (a) and (b) show the results for data formats of NetCDF4 and PNetCDF with CDF5 features respectively. Columns of different colors shared by both figures represent different numbers of I/O tasks configured to read or write a file simultaneously.

TABLE 2
Optimization Results of Initialization Decomposition for Variable *qv*

I/O Module/MPI Ranks	30,000	60,000	120,000
Original I/O Module	76.6s	310.7s	576.6s
Refactored I/O Module	0.11s	0.054s	0.027s

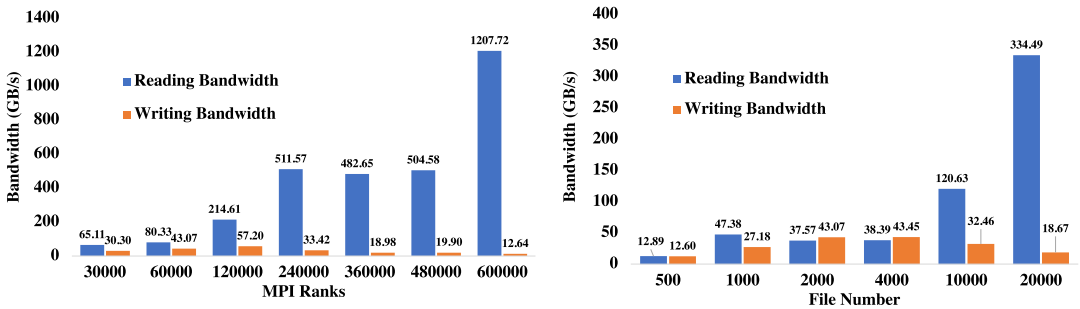
same I/O stride for different scales, so the number of files for both input and output is the outcome of dividing the scale by the value of the I/O stride and the size of each sub-communicator is the value of I/O stride. Also, only the processes with local rank 0 (MPI rank in sub-communicator) are responsible for I/O to avoid manually tuning for the I/O tasks' number per file. The experimental results are shown in Fig. 10a. With a reading bandwidth of 1207.72 GB/s at scale of a 600,000 MPI ranks, 20,000 input files of the total size of 1.1T can be entirely read in about 0.9 seconds. Writing performance is worse than reading, but on a scale of 600,000, it is still possible to write a checkpoint of 3.3T in 267.4 seconds with a bandwidth of 12.64 GB/s. Despite the refactored I/O takes extra time to create files, multiple-file I/O is at least dozens of times faster than the original I/O module. Accordingly, we can conclude that our refactored I/O module is scalable and efficient enough to deal with I/O problems at an ultra-large scale.

We also test the impact of file number on I/O performance by changing the I/O stride on the scale of 60,000 MPI ranks, shown in Fig. 10b. The general trend of input performance increases with the number of files, but performance is similar for 1000, 2000, and 4000 files, which may be affected by the file system. Output performance rises initially but then drops when the parallel file system becomes the bottleneck for using too many files. When increasing file number, the size of each communicator decreases. When setting the I/O stride to 3 for the scale of 60,000, the program reads and writes 20,000 files, respectively, but each communicator contains only 3 MPI processes, while there are 30 MPI processes for the scale of 600,000. The larger the scale, the more I/O nodes can be allocated, and the theoretical I/O performance should be better. For the example of using 20,000 files, the scale of 60,000 processes has a read bandwidth of 334.49 GB/s, which is 1/4 of the bandwidth of 600,000 processes. However, the reading performance under 60,000 processes is higher than the theoretical I/O performance (60000/1024*4GB/s = 234.375GB/s), probably because compute nodes are assigned across more I/O nodes.

5.2 Kernel Speedup

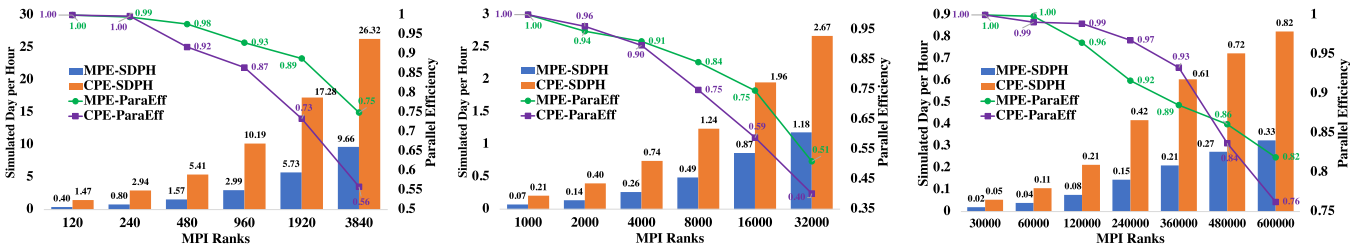
Fig. 12 shows the speedup for different optimizations, in which the blue columns marked as CPE represent the speedup after being parallelized with the scheme discussed in Section 4.3, compared with the MPE-only codes, and the orange columns are further accelerated with the hybrid buffering scheme (HBS) mentioned in Section 4.5. The speedup for tracer transport redesign (in Section 4.4) is also reported, and the column is coloured grey. For the hardware configuration, part of LDM (32KB) is configured as a hardware cache, and the remaining 224KB space is utilized to allocate local variables or manually control swap-in and swap-out. If more space is used as the hardware cache, more local temporary variables must be allocated in the main memory, increasing the data swapping overhead, especially for physics and chemistry parameterizations.

Fig. 12 demonstrates that *rrtmg_lw* and *rrtmg_sw* achieve 4.5× and 7.2× speedup, respectively. The worst optimized kernel is acoustic in the dynamic core, which has only 1.5× improvement, mainly constrained by the high proportion of communication. *Chem_trans_2* has achieved a speedup of 7.1× after running on CPE, then additionally improved by 2.7× and 1.03× with HBS and the redesigned algorithm, respectively. *chem_trans_1* obtains a 9.7× speedup. *phy_trans_1* and *phy_trans_2* are similar to the corresponding parts in chemistry, and they obtain total speedups of 19.1× and 10.2×, respectively. *phy_trans_2* only performs communication aggregation without the overlap of computation and communication because the number of loops that iterate over the tracer is relatively small. Other computational procedures in the dynamic core, accounting for a relatively small percentage, achieve good performance improvements, such as *dyn_tend* of a 10.5× speedup, *rec_state* of a 16× speedup and *solve_diag* of a 14.8× speedup. In addition, Fig. 12 also shows the contribution of hybrid buffering scheme to accelerate part of procedures in the dynamic core, which are improved from 2.7× to 6.3×. The chemical procedures achieve 3.5×, 6.5×, and 13.3× improvements after porting on CPEs for *wetscav*, *drydep*, and *optical*.



(a) Impact of MPI Ranks (b) Impact of File Number

Fig. 10. (a) I/O bandwidth is tested by scaling MPI ranks from 30,000 to 600,000 with a fixed I/O stride of 30. (b) I/O bandwidth is tested using the same MPI ranks of 60,000 but changing total file numbers from 500 to 20,000.



(a) U60KM (b) V4KM (c) U3KM

Fig. 11. Strong scaling of MPAS-A for different problem sizes. Parallel efficiency is 40% when using 32,000 MPI processes because the cells' number assigned to each CG is only 24 in horizontal and 56 in vertical (total cell number is calculated by multiplying 24 and 56, and in this work the vertical layer number is fixed for U60KM, V4KM, and U3KM), which is too small to exploit the computing power of CPE fully. Similarly, cells' horizontal number in each CG is 42 for U60KM at the scale of 3840, resulting in 56% parallel efficiency. Problem size of U3KM has a parallel efficiency of 0.76 with an average cell number of 109 in horizontal.

5.3 Strong Scaling

As shown in Fig. 11a, with the number of MPI processes (or CGs) increasing from 120 to 3840 for U60KM simulations, we can observe good performance benefits from CPE acceleration. The fastest simulation speed achieved for U60KM is 26.3 SDPH when using 3840 MPI processes in our test. In Fig. 11b, we show the performance results for V4KM simulations. 2.67 SDPH has been achieved at 32,000 MPI processes. Due to the computation pattern of MPAS-A, no matter what size the problem is, the increase in scale decreases the number of cells assigned to each MPI process. When the scale is large, the workload per CG is small, resulting in a low computational share and the CPE's computational capability not being fully utilized. For example, the parallel efficiency of V4KM is 40% on a scale of 32,000 MPI ranks because the average number of cells assigned to each MPI process is only 24 in horizontal and 56 in vertical (the total number should be obtained by multiplying 24 and 56, and in this work, the vertical layer is fixed, so we only mention the cell number in horizontal). Similarly, for U60KM at the scale of 3840, the average number of cells in each MPI process is 42 in horizontal. In addition, the number and size of messages sent by an MPI task to its neighbours are roughly constant, but more MPI processes are involved in communication at a larger scale, resulting in more communication overhead. We then test the strong scalability of MPAS-A at U3KM by enlarging the scale from 30,000 to 600,000. As shown in Fig. 11c, the performance evaluated in SDPH improves from 0.05 to 0.82. At the largest scale, MPAS-A has a parallel efficiency of 82% on MPEs only and 76% after using CPEs. For 600,000 MPI ranks, each MPI process has an average horizontal cell number of 109 and compared with the MPE-only version, MPAS-A has achieved an improvement of 2.54× after being ported and optimized on CPEs.

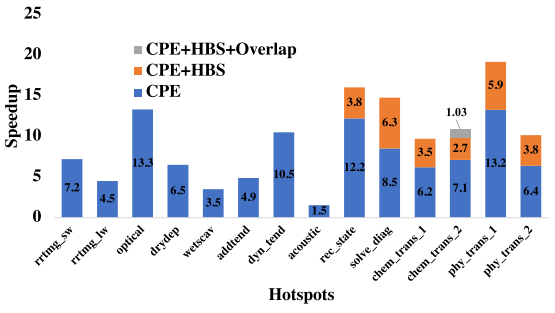


Fig. 12. Speedup of different hotspots. The blue columns labelled by the *CPE* represent the speedup of using CPE. The orange columns labelled by *CPE+HBS* are the results of additionally using the hybrid buffering scheme. The grey column is the speedup for overlapping the computation and communication. The baseline is the MPE-only version without any optimization, and I/O optimization does not affect the execution time of hotspots.

5.4 Weak Scaling

With weak scaling, the number of cells processed per CG remains constant as the system size increases, so ideally, the execution speed of a single time step of the program also remains constant. For MPAS-A, to increase the resolution, one must shorten the time step to ensure the stability of the simulation. As a result, MPAS-A can achieve weak scaling from the aspect of time-per-time step rather than from the time-to-solution perspective [45]. In other words, when we simulate the problem size of U3KM with a time step of 15 seconds, we need to choose the problem size of U60KM with a 300 seconds time step as the base and change the number of cells assigned to each CG. Fig. 13 shows weak scaling efficiency for time-per-time step, dividing results from the larger U3KM problem size by the smaller U60KM problem size for a given amount of work. The results indicate that MPAS-A provides good weak scaling. The efficiency of the MPE-only version consistently remains above 97% as the number of cells processed per CG increases, and the cases of using CPE maintain efficiency above 90%.

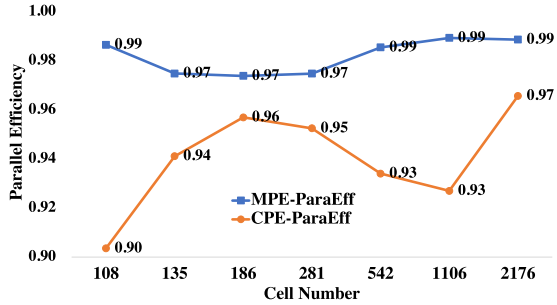


Fig. 13. Weak scaling of MPAS-A. The weak scaling is tested from the aspect of time-per-time step instead of time-to-solution by changing cell numbers. We choose the problem size of U60KM with a time step of 300s as base and test the weak scaling for U3KM with a time step of 15s.

6 Conclusion

The problem of two-phase I/O has been extensively studied, but I/O is still a significant impediment to atmospheric applications' scalability and simulation efficiency because the I/O modules of these applications have not been well designed. In addition, some of the I/O methods currently used in applications still have limitations in front of the extreme scale. In this paper, we first elaborate on the causes of the severe I/O bottleneck known as two-stage I/O for MPAS-A and then convert its I/O mode from N-to-1 to N-to-M by rearranging the file data to enable high-performance file system access and low-cost communication. By scaling MPAS-A with I/O to 600,000 MPI processes on the new Sunway supercomputer, we demonstrate the necessity and efficiency of our method. Moreover, we have conducted methods including tracer transport redesign and a hybrid buffering scheme to accelerate the computation. With the three-level parallelization scheme, 39 million heterogeneous cores are utilized on 600,000 MPI ranks, achieving a parallel efficiency of 76%.

Our future works include 1) Communication optimization. Halo exchanges account for the highest percentage besides the computation during the time integration, but we cannot optimize most of them using the same method as tracer transport. 2) There is still room for the acceleration of computation parts. As discussed in Section 3.4.1, SIMD units must be further considered to maximize the computing power of the SW26010pro, which requires improving data reuse and reducing or hiding the overhead of data transfer and copying.

References

[1] W. C. Skamarock et al., "A description of the advanced research WRF model version 4," *Nat. Center Atmospheric Res.*, vol. 145, p. 145, 2019.

[2] W. C. Skamarock, J. B. Klemp, M. G. Duda, L. D. Fowler, S.-H. Park, and T. D. Ringler, "A multiscale nonhydrostatic atmospheric model using centroidal voronoi tesselations and c-grid staggering," *Monthly Weather Rev.*, vol. 140, no. 9, pp. 3090–3105, 2012.

[3] R. B. Neale et al., "Description of the NCAR community atmosphere model (cam 5.0)," *NCAR Tech. Note NCAR/TN-486+ STR*, vol. 1, no. 1, pp. 1–12, 2010.

[4] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Multi-core acceleration of chemical kinetics for simulation and prediction," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, 2009, pp. 1–11.

[5] J. Y. Kim, J.-S. Kang, and M. Joh, "GPU acceleration of MPAS microphysics WSM6 using openacc directives: Performance and verification," *Comput. Geosci.*, vol. 146, 2021, Art. no. 104627.

[6] I. Carpenter et al., "Progress towards accelerating homme on hybrid multi-core systems," *Int. J. High Perform. Comput. Appl.*, vol. 27, no. 3, pp. 335–347, 2013.

[7] C. Yang et al., "10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2016, pp. 57–68.

[8] K. Xu et al., "Refactoring and optimizing WRF model on sunway taihulight," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.

[9] H. Fu et al., "The sunway taihulight supercomputer: System and applications," *Sci. China Informat. Sci.*, vol. 59, no. 7, pp. 1–16, 2016.

[10] J. Gao et al., "Sunway supercomputer architecture towards exascale computing: Analysis and practice," *Sci. China Informat. Sci.*, vol. 64, no. 4, pp. 1–21, 2021.

[11] H. Shang et al., "Extreme-scale AB initio quantum raman spectra simulations on the leadership HPC system in china," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–13.

[12] Y. Liu et al., "Closing the "quantum supremacy" gap: Achieving real-time simulation of a random quantum circuit using a new sunway supercomputer," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–12.

[13] J. Xiao et al., "Symplectic structure-preserving particle-in-cell whole-volume simulation of tokamak plasmas to 111.3 trillion particles and 25.7 billion grids," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–13.

[14] Q. Zhu, H. Luo, C. Yang, M. Ding, W. Yin, and X. Yuan, "Enabling and scaling the HPCG benchmark on the newest generation sunway supercomputer with 42 million heterogeneous cores," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–13.

[15] D. Heinzeller, M. G. Duda, and H. Kunstmann, "Towards convection-resolving, global atmospheric simulations with the model for prediction across scales (MPAS) v3. 1: An extreme scaling experiment," *Geoscientific Model Develop.*, vol. 9, no. 1, pp. 77–110, 2016.

[16] The HDF group, "Hierarchical data format," 2022. [Online]. Available: https://www.hdfgroup.org/HDF5/

[17] R. Rew and G. Davis, "NetCDF: An interface for scientific data access," *IEEE Comput. Graph. Appl.*, vol. 10, no. 4, pp. 76–82, Jul. 1990.

[18] J. Li et al., "Parallel netCDF: A high-performance scientific I/O interface," in *Proc. ACM/IEEE Conf. SuperComput.*, 2003, pp. 39–39.

[19] A. Geist et al., "MPI-2: Extending the message-passing interface," in *Proc. Eur. Conf. Parallel Process.*, 1996, pp. 128–135.

[20] W. F. Godoy et al., "Adios 2: The adaptable input output system. A framework for high-performance data management," *SoftwareX*, vol. 12, 2020, Art. no. 100561.

[21] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (adios)," in *Proc. 6th Int. Workshop Challenges Large Appl. Distrib. Environ.*, 2008, pp. 15–24.

[22] J. Edwards, J. M. Dennis, M. Vertenstein, and E. Hartnett, Parallel I/O libraries (PIO), 2022. [Online]. Available: https://ncar.github.io/ParallelIO

[23] Software for caching output and reads for parallel I/O (scorpio), 2022. [Online]. Available: https://e3sm.org/scorpio-parallel-io-library/

[24] B. Yang et al., "End-to-end I/O monitoring on a leading supercomputer," in *Proc. 16th {USENIX} Symp. Netw. Syst. Des. Implementation*, 2019, pp. 379–394.

[25] Q. Chen, K. Chen, Z.-N. Chen, W. Xue, X. Ji, and B. Yang, "Lessons learned from optimizing the sunway storage system for higher application I/O performance," *J. Comput. Sci. Technol.*, vol. 35, no. 1, pp. 47–60, 2020.

[26] T. Balle and P. Johnsen, "Improving I/O performance of the weather research and forecast (WRF) model," *Cray User Group*, 2016.

[27] B. Hadri, "Introduction to parallel i/o," 2011. [Online]. Available: https://www.nics.tennessee.edu/files/pdf/hpcss13_14/04_08_Parallel_IO_Part1.pdf

[28] S. Mendez, S. Lührs, D. Sloan-Murphy, A. Turner, and V. Weinberg, "Best practice guide-parallel I/O," 2019.

[29] J.-S. Kang, H. Myung, and J.-H. Yuk, "Examination of computational performance and potential applications of a global numerical weather prediction model MPAS using kisti supercomputer nurion," *J. Mar. Sci. Eng.*, vol. 9, no. 10, 2021, Art. no. 1147.

[30] Q. Kang et al., "Improving all-to-many personalized communication in two-phase I/O," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–13.

[31] Q. Kang, S. Breitenfeld, K. Hou, W.-K. Liao, R. Ross, and S. Byna, "Optimizing performance of parallel I/O accesses to non-contiguous blocks in multiple array variables," in *Proc. IEEE Int. Conf. Big Data*, 2021, pp. 98–108.

[32] E3sm. [Online]. Available: http:https://e3sm.org/model

[33] H. Yashiro et al., "A 1024-member ensemble data assimilation with 3.5-km mesh global weather simulations," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–10.

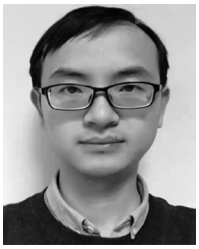
[34] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, Top 500 supercomputer lists. [Online]. Available: http://www.top500.org

[35] H. Fu et al., "Refactoring and optimizing the community atmosphere model (CAM) on the sunway taihulight supercomputer," in *Proc. Proc. Int. Conf. for High Perform. Comput., Netw., Storage Anal.*, 2016, pp. 969–980.

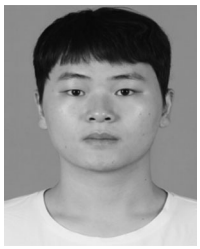
[36] H. Fu et al., "Redesigning cam-se for peta-scale climate modeling performance and ultra-high resolution on sunway taihulight," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2017, pp. 1–12.
[37] L. J. Wicker and W. C. Skamarock, "Time-splitting methods for elastic models using forward time schemes," *Monthly Weather Rev.*, vol. 130, no. 8, pp. 2088–2097, 2002.
[38] C. Zhao et al., "Exploring the impacts of physics and resolution on aqua-planet simulations from a nonhydrostatic global variable-resolution modeling framework," *J. Adv. Model. Earth Syst.*, vol. 8, no. 4, pp. 1751–1768, 2016.
[39] F. Judt, "Insights into atmospheric predictability through global convection-permitting model simulations," *J. Atmospheric Sci.*, vol. 75, no. 5, pp. 1477–1497, 2018.
[40] J. Gu et al., "Establishing a non-hydrostatic global atmospheric modeling system at 3-km horizontal resolution with aerosol feedbacks on the sunway supercomputer of china," *Sci. Bull.*, vol. 67, pp. 1170–1181, 2022.
[41] J. Dongarra, "Report on the sunway taihulight system," Univ. Tennessee, Oak Ridge Nat. Lab., Oak Ridge, TN, USA. Accssed: June, 20, 2016, www.netlib.org
[42] Z. Ma et al., "Bagualu: Targeting brain scale pretrained models with over 37 million cores," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 192–204.
[43] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.
[44] G. A. Grell et al., "Fully coupled "online" chemistry within the wrf model," *Atmospheric Environ.*, vol. 39, no. 37, pp. 6957–6975, 2005.
[45] J. Michalakes et al., "AVEC report: NGGPs level-1 benchmarks and software evaluation," in *Proc. Adv. Comput. Eval. Committee*, 2015, p. 22.

Xiaoyu Hao is currently working toward the PhD degree with the Advanced Computer System and Architecture (ACSA) Laboratory, Department of Computer Science and Technology, University of Science and Technology of China. His research interest is high performance computing for scientific applications on large scale systems.

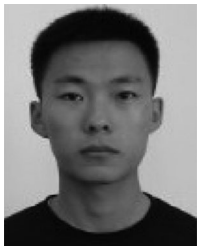
Tao Fang received the MS degree in computer science from the University of Science and Technology of China in 2021. His research interests include performance tuning of scientific applications on domestic supercomputer and cloud computing service.



Junshi Chen received the PhD degree in computer science from the University of Science and Technology of China (USTC), Hefei, in 2020. He is a postdoctor with USTC. His research interests include highperformance computing and computer architecture.



Jun Gu received the BS degree in atmospheric science from the University of Science and Technology of China in 2020. He is currently working toward the master's degree with the University of Science and Technology of China. His research interest focuses on developing atmospheric models and apply these models to conduct high-resolution simulation.



Jiawang Feng received the BS degree in atmospheric science from the University of Science and Technology of China in 2018. He is currently working toward the PhD degree with the University of Science and Technology of China. His research focuses on developing atmospheric models and apply these models to investigate atmospheric scientific problems related to air quality and aerosol effects.

Hong An received the PhD degree in computer science from the USTC in 2000. She is currently a professor with the School of Computer Science and Technology, University of Science and Technology of China (USTC). She is the director with the Advanced Computer System Architecture (ACSA) Lab in the USTC. Her main research focuses on parallel computer architecture, parallel programming, operating system design, and high performance computing.

Chun Zhao received the PhD degree in atmospheric science from the Georgia Institute of Technology, U.S.A, in 2009. He is currently a professor with the University of Science and Technology of China. His research focuses on applying advanced computing methods to develop atmospheric models and apply these models to investigate atmospheric scientific problems related to air quality, extreme weather event, climate change, and planetary atmospheric environment.