Computers, Materials & Continua CMC, vol.63, no.3, pp.1399-1413, 2020

Parallelization and I/O Performance Optimization of a Global Nonhydrostatic Dynamical Core Using MPI

Tiejun Wang[1], Liu Zhuang[2], Julian M. Kunkel[3], Shu Xiao[1] and Changming Zhao[1, *]

Abstract: The Global-Regional Integrated forecast System (GRIST) is the next-generation weather and climate integrated model dynamic framework developed by Chinese Academy of Meteorological Sciences. In this paper, we present several changes made to the global nonhydrostatic dynamical (GND) core, which is part of the ongoing prototype of GRIST. The changes leveraging MPI and PnetCDF techniques were targeted at the parallelization and performance optimization to the original serial GND core. Meanwhile, some sophisticated data structures and interfaces were designed to adjust flexibly the size of boundary and halo domains according to the variable accuracy in parallel context. In addition, the I/O performance of PnetCDF decreases as the number of MPI processes increases in our experimental environment. Especially when the number exceeds 6000, it caused system-wide outages (SWO). Thus, a grouping solution was proposed to overcome that issue. Several experiments were carried out on the supercomputing platform based on Intel x86 CPUs in the National Supercomputing Center in Wuxi. The results demonstrated that the parallel GND core based on grouping solution achieves good strong scalability and improves the performance significantly, as well as avoiding the SWOs.

Keywords: MPI, parallelization, performance optimization, global nonhydrostatic dynamical core.

1 Introduction

Nowadays, several dynamical core models, such as DYNAMICO [Dubos, Dubey, Tort et al. (2015)], FVM [Smolarkiewicz, Kühnlein and Grabowski (2017)], GEM [Girard, Plante, Desgagné et al. (2014)] and ICON [Zängl, Reinert, Rípodas et al. (2015)], have been developed as a fundamental component of global atmospheric modeling systems. The study of dynamical core models has become increasingly important for both numerical weather prediction and climate studies. The Global-Regional Integrated forecast System (GRIST) is the next-generation weather and climate integrated model dynamic framework and aiming

[1] School of Computer Science, Chengdu University of Information Technology, Chengdu, 610225, China.

[2] National Supercomputing Center in Wuxi, Wuxi, 214072, China.

[3] Department of Computer Science, University of Reading, Berkshire, RG6 6UR, UK.

[*]Corresponding Author: Changming Zhao. Email: zcm84@cuit.edu.cn.

Received: 18 January 2020; Accepted: 28 February 2020.

to extend to be a fully-fledged atmospheric general circulation model [Yu, Zhang, Wang et al. (2019)]. A prototype of GRIST is being developed by Chinese Academy of Meteorological Sciences and composed of two parts: one part is a global shallow water modeling (SWM) framework [Zhang (2018)], which was developed on an unstructured Voronoi-Delaunay grid and tested against various two-dimensional (2D) benchmarks by isolating most horizontal components of a 3D model; the other part is a new global nonhydrostatic dynamical (GND) core [Zhang, Li, Yu et al. (2019)] for supporting multiscale modeling of the atmosphere and enabling resolve more fine-scale fluid structures without a uniform increase in the global resolution. The GND core is based on unstructured mesh which allows a flexible switch between the quasi-uniform and VR configurations by altering the underlying mesh descriptor information. In this paper, we focus on the parallelization and performance optimization to GND core as the simulation to 3D model is more significant than 2D model in a weather and climate system.

This paper is structured as follows: Section 2 discusses the background of our research work, then Sections 3 and 4 describe the parallel GND core implementation, Section 5 shows the experiments environment and analyze the performance results. Finally, the paper is summarized in Section 6.

2 Background

2.1 The original serial GND core

Normally, running the original serial GND core is a sequence of three parts: reading, computing and writing. Firstly, all the one-dimension and two-dimension data describing the grid structure and other initial information stored in data files are read into the memory of computing nodes to initialize the model. Then, the model is run step by step according to the initial information to simulate the atmospheric motion to produce predictions on the computing platform. Finally, all the predictions in memory are dumped out to storage as data files which could be used to support further computing.

Table 1: Time consumption in serial running for different resolution G7 and G8

Resolution	Average Reading Time(s)	Average Writing Time(s)	Average Computing Time(s)
G7	1.1	6.8	5220.2
G8	2.4	27.1	31335.5

The main goal of optimization is to reduce the running time as well as ensuring results correct. Analyzing the running procedure, massive I/O operations are involved in the first and third parts where huge amount of data will be read from and write to data files with parallel netCDF (PnetCDF) [Gao, Liao, Choudhary et al. (2009)], and considerable computing operations are included in the second part to simulate the atmospheric motion in 24 hours using the hydrostatic solver (HDC) or the nonhydrostatic solver (NDC). The original serial model could be run in single node with 2 Intel CPUs and 128 G memory. Analyzing the running results as shown in Tab. 1, the main body of time consumption is the computing time where the reading and writing time for I/O operations only accounts for a negligible proportion. Furthermore, the simulation for problem size greater than G8 is not runnable due to the memory limitation of single node. In order to reduce the total

running time and improve the ability to deal with high resolutions, we applied the efficient large-scale parallel computing based on the latest high-performance computing platform to optimize the GND core.

2.2 Unstructured mesh

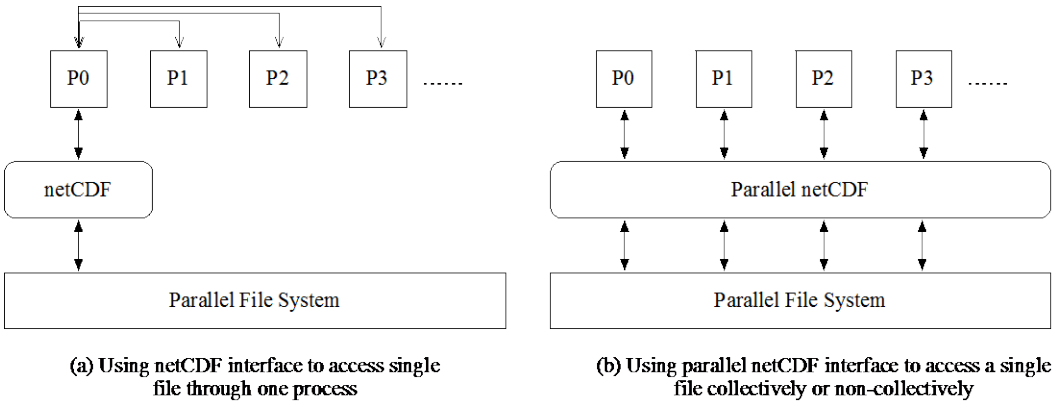
The GND core is based on unstructured mesh which allows a flexible switch between the quasi-uniform and variable-resolution (VR) configurations by altering the underlying mesh descriptor information. Edge points, triangle points and hexagon points are three kinds of points on the GND core mesh, where each point can perceive the nearest neighbors. The mesh is essentially the unstructured Voronoi-Delaunay grid. At the beginning, quasi-uniformly distributed points are generated on the sphere as the typical icosahedral hexagonal-pentagonal (IHP) shape. Changing the mesh descriptor information could modify the distribution of triangle and hexagon points on the unstructured mesh to affect the mesh resolution.

One-level finer grids are generated by bisecting each triangular edge of the former coarser grid. The generated resolution of this mesh is referred to as grid level G*n*, where *n* denotes the number of bisections. In the implement of GND core, the centroidal Voronoi constraint method [Du, Gunzburger and Ju (2003); Ringler, Ju and Gunzburger (2008)] is used to optimize the subdivided IHP mesh. As the scale of problem size increases, the time and memory consumption will escalate so that the mesh resolution that could be handled in single node is limited by the hardware resources. Fortunately, parallelization can not only reduce the total running time, but also increase the scale of problems that system could deal with.

2.3 NetCDF and PnetCDF

Network common data format (netCDF) is widely used in the field of meteorological research and applications for storing and retrieving data in the form of arrays. Actually, it is a library of data access functions which support a view of data as a collection of self-describing and portable objects. Starting with version 4.0, the netCDF API supports the use of the HDF5 data format [Folk, Heber, Koziol et al. (2011)], which allows netCDF users to create HDF5 files with benefits that are not available with the netCDF format, such as much larger files and multiple unlimited dimensions. Unfortunately, as lacking parallel access mechanism in the original design of netCDF interface, the capability of providing services to parallel applications is significantly limited. In particular, concurrently writing to a netCDF file (NC file) is not supported. Therefore, serial accessing to a NC file through only one of multiple processes could easily become a performance bottleneck as show in Fig. 1(a).

Li et al. introduced a parallel interface for writing and reading data stored in NC files [Li, Liao, Choudhary et al. (2003)], and PnetCDF interface improves the parallel I/O performance significantly as well as keeps the convenience with minimal changes to original netCDF interface as shown in Fig. 1(b). MPI-IO is used in PnetCDF interface to achieve the parallel I/O features. Migrating the GND core implementation from netCDF to PnetCDF could provide the capability of accessing single dataset with multiple processes concurrently.



(a) Using netCDF interface to access single file through one process

(b) Using parallel netCDF interface to access a single file collectively or non-collectively

Figure 1: Accessing single NC file through netCDF interface and PnetCDF interface in parallel applications

3 Parallel implement of GND core

3.1 Domain decomposition

The main objective of parallelization is to allow multiple processes to compute simultaneously while ensuring the results correct. Many applications in parallel computing use *domain decomposition* to distribute the computing tasks among different processing elements.

In this paper, we adopted METIS [LaSalle, Patwary, Satish et al. (2015)] for graph partitioning to obtain a domain decomposition leading to a good balancing of both the size of domain interiors and the size of interfaces, which was a key point for load balancing and efficiency in a parallel context. The result of partitioning was that the whole computing tasks with associated data were split into multiple cells adjacent to each other. We used home cells (see Fig. 2(b)) to refer the cells those associated with specific MPI processes to perform computing. A local stencil was composed of a home cell and other cells surrounding it as shown in Fig. 2(a). Those home cells, also known as computation domains, were composed of an inner domain (light gray region) and a boundary domain (dark gray regions). The halo domain essentially referred to the overlapping regions (the stippled regions in Fig. 2(a)), which referred to the boundary domains of other cells in local stencils. Combining computation domain and halo domain formed a full domain (see Fig. 2(c)). In fact, when the partitioning using METIS was complete, computation domains were already known to specific MPI processes, but halo domains not. Therefore, only after obtaining the full domain could an MPI process start the computing task, which meant that the halo domain had to be updated from neighbor cells in the local stencil. The operation that a home cell exchanged its boundary domains with neighbors was called the neighbor exchange or halo exchange. As shown in Fig. 2(c), halo (1) and halo (2) regions would exchange data with boundary (1) and boundary (2) regions, respectively.

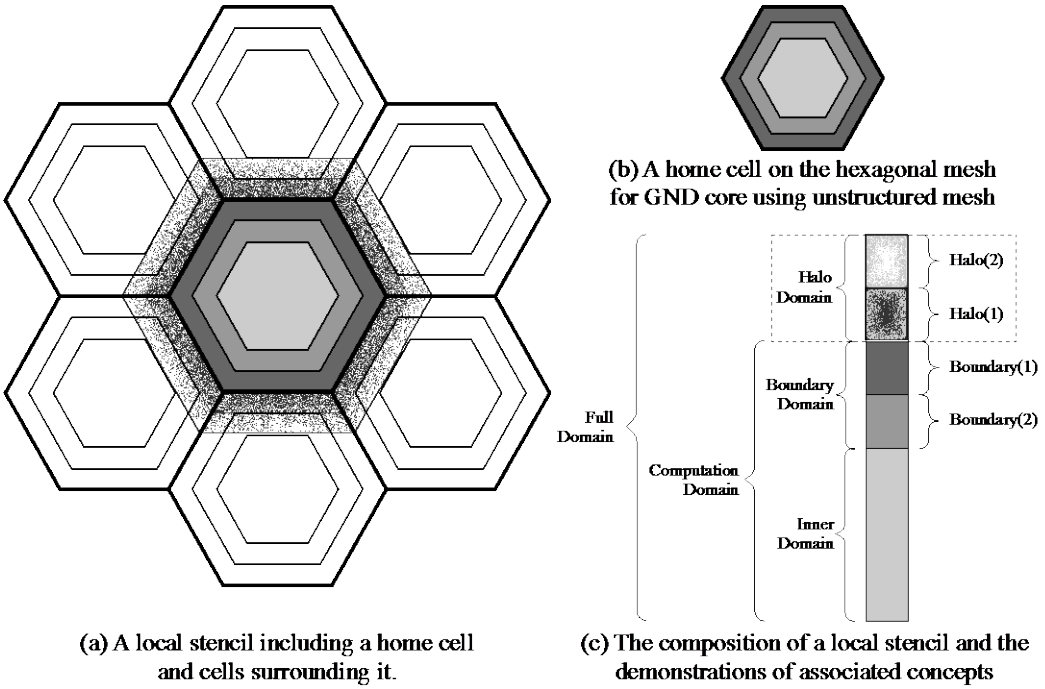


Figure 2: An illustration of a local stencil on the hexagonal mesh of GND core

3.2 Computing performance improvement

Increasing the numbers of halo and boundary regions could improve the calculation accuracy, but at the same time it will also increase the load of communication and calculation. To isolate the implement of communication module, a new structure linked list was designed as shown in Fig. 3(a), which allows to flexibly adjust the numbers of halo and boundary regions according to accuracy requirements without changing to the communication module. In order to balance the calculation accuracy with communication and calculation load, the numbers of halo and boundary regions are all set to 2 as shown in Fig. 2(c).

In fact, there were two mature techniques, OpenMP and MPI, used to improve computing performance, and MPI was chosen finally. On the one hand, MPI was required for the purpose of the GND core running on a cluster platform. On the other hand, reprogramming of the hotspot functions using OpenMP only obtained little improvement. Furthermore, according to the computing logic, the whole computing task for computation domain was split into two separate parts: inner domain computing and boundary domain computing. The asynchronous communication functions of MPI were adopted to synchronize the computing for inner domain with the halo exchange between boundary domains and halo domains, and which further increased parallel scalability. In order to facilitate the separation of computing and communication, a set of communication interfaces was also designed as shown in Fig. 3(b), where interface *exchange_data* encapsulated *MPI_Isend* and *MPI_Irecv* functions to exchange data with other processes. Combining multiple data fragments *data_i* into an *exchange_field_list*

typed linked list *field_head* could not only reduce the numbers of communication requests, but also improve the communication efficiency.

```
type exchange_field_list
    type(scalar_field),  pointer :: field_data
    type(exchange_field_list), pointer :: next => null()
end type exchange_field_list
```

(a) Structure linked list to flexibly adjust the number of exchanged regions

```
exchange_data_add(mesh, field_head, data_1)
exchange_data_add(mesh, field_head, data_2)
                  ⋮                              } Data Preparation
exchange_data_add(mesh, field_head, data_n)

exchange_data(mesh,field_head) ← Communication Integration
```

(b) Communication interfaces to separate computing and communication

Figure 3: Data structure and interfaces designed for halo exchanges

Moreover, according to the principle of data locality, the data was stored in two-dimensional $K \times M$ mode, where K was the layer number of the mesh in the vertical direction and M was the points number in computation domain. That data storage mode guaranteed that the data in the vertical direction was continuously arranged in the memory, which could take full advantage of the vectorization of CPU, improve the cache utilization and reduce time consumption of data processing.

Meanwhile, we reconstructed some data structures of GND core to improve the performance. For example, halo, inner and boundary domains were abstracted into *global_domain* struct to store pointers to other parts and information of triangle and polygon (hexagon in this paper) together with those vertices and edges. As shown in Fig. 4, original source (left part) adopted *vertice_structure* struct variable *vtx* to calculate divergence where variable *vtx* belonging to *global_domain* struct variable mesh included two members: *nnb* (number of neighbor nodes) and *ed* (edges). We converted the indirect access to the members *nnb* and *ed* of struct *vtx* to the direct access to a one-dimension array *vtx_nnb* and a two-dimension array *vtx_ed* (right part), which reduced the overhead of indirect indexing.

```
do inb = 1, mesh%vtx(iv)%nnb                  do inb = 1, mesh%vtx_nnb(iv)
   index_edge = mesh%vtx(iv)%ed(inb)    ⇒     index_edge = mesh%vtx_ed(inb, iv)
        ⋮                                             ⋮
```

Figure 4: Reconstructing hotspot code

4 Parallel I/O optimization

In terms of parallel I/O performance, we replaced original netCDF with PnetCDF library. Then, all the MPI processes could read and write the same NC file simultaneously, which improved the read and write efficiency as well as reducing memory usage. Unfortunately,

current PnetCDF does not provide functionality for reading or writing multiple array variables in a single call, so this limitation may reduce the I/O performance for accessing a large number of small-sized array variables [Gao, Liao, Choudhary et al. (2009)]. The following experiment confirmed the speculation about that issue. We designed two data reading solutions: the non-grouping solution and the grouping solution. In the non-grouping solution, all the MPI processes accessed the same NC file of size 2 G directly through the PnetCDF interface. In the grouping solution, only the host process (process number is 0 in the group) was allowed to read the NC file through PnetCDF interface, and all the other guest processes sent requests to and got data from the host process. The experimental results shown in Tab. 2 revealed the fact that the time consumption of the grouping solution was much less than that of the non-grouping solution.

Table 2: Comparation of experimental results

Number of Processes	**1**	**24**	**48**	**72**	**96**	**110**	**150**
Number of Nodes	1	1	2	3	4	5	7
Time consumption of non-grouping (s)	1.96	9.80	17.85	21.12	21.80	21.33	23.95
Time consumption of grouping (s)		1.80	3.02	3.35	3.42	3.43	3.45

Furthermore, according to our experimental results, the reading and writing performance dropped dramatically when the number of processes exceeded 600. In particular, system-wide outages (SWO) occurred as calling two functions *nfmpi_iget_vara_double* and *nfmpi_iput_vara_double* with PnetCDF running in non-blocking mode when the number of MPI processes was more than 6000. In order to overcome SWOs, a grouping parallel I/O solution was proposed in this paper. Although some overhead would be added and thereby increasing the running time, the overhead could be negligible when the scale of the problem reached a certain level. The grouping solution and associated concepts could be described as follows.

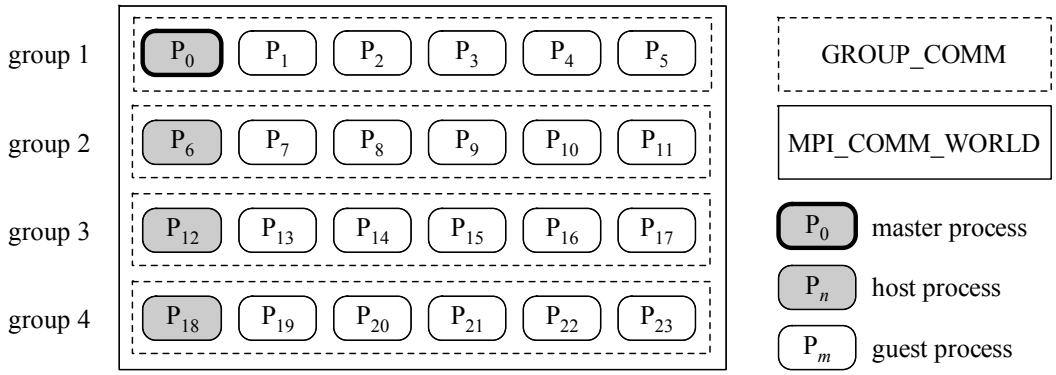


Figure 5: Reconstructing hotspot code

Let c be the group size and i be the index of MPI process P_i in global communication group MPI_COMM_WORLD, then the index j of MPI process P_i in group g could be defined as Eq. (1) and the process with an index equal to 0 in group would be the *host process* of that group.

$$g = i / c$$
$$j = i \bmod c \tag{1}$$

Fig. 5 shows an illustration of the grouping solution where there are 24 MPI processes in communication group MPI_COMM_WORLD and P_0 is the master process. In addition, there are 6 MPI processes in each communication group and P_0, P_6, P_{12} and P_{18} is the host process of groups 1, 2, 3 and 4, respectively. In the grouping solution, only 4 host process of each group accesses the NC file and other guest processes would send the access requests to and get requested data from host process. Although this grouping solution could effectively reduce the concurrent access to NC file, it introduces the overhead to gather requests and scatter data obtained from NC files.

5 Performance evaluation

To evaluate the performance and scalability of our parallel GND core, we designed some experiments and compared the results with the original serial GND core. We also designed a group of experiments with different group sizes in term of the influence of the group size on the performance. The original GND core was developed in Fortran 90 language and compiled by Intel Fortran Composer XE for Linux 2013 update 4 with -O2 optimization option.

The experiments were run on the commercial auxiliary computing system of national supercomputing center in Wuxi. This system is a petaflop-scale cluster with 980 compute nodes. Each compute node has 128 GB of memory shared among its two 2.5 GHz Intel Xeon E5-2680 v3 processors and each CPU has 12 cores. All the compute nodes are interconnected by switches and also connected via switches to the multiple I/O nodes running the GPFS parallel file system. The aggregate disk space is 15 PB and the peak I/O bandwidth is 14 GB/s. In all the experiments, each MPI process was mapped into a physical CPU core at runtime, which meant the fact that the maximum number of processes a single computing node allowed to run is 24. The resolutions in this paper include G7 (~60 km; 163,842 cells), G8 (~30 km; 655,362 cells) and G9 (~15 km; 2,621,442 cells) as shown in Tab. 3.

Table 3: The parameters and resolution used in different grid levels

Grid Level	G7	G8	G9
Cells Number	163,842	655,362	2,621,442
Iteration Steps	288	432	864
Resolution	~60 km	~30 km	~15 km
NC File Size	289.7 MB	1.16 GB	4.63 GB

5.1 Strong scaling analysis of non-grouping GND core

We applied all the methods mentioned in Sections 3 and 4 to implement an optimized GND core supporting parallelization developed in Fortran 90 language. In order to distinguish from the system obtained by only using the methods descripted in Section 3 to optimize I/O performance, *parallel GND core* is used here to refer to the former and *non-grouping GND core* refers to the latter.

Due to the limitation of the memory size (128 GB) in computing nodes, original serial GND core could not handle the problem with resolution higher than G8. Therefore, we ran the serial GND core for three times to deal with the same data set stored in NC files for different resolutions G7 and G8, respectively. For comparison purpose, we ran the parallel GND core for three times to handle the same NC files as well.

Fig. 6 (left) shows the strong scaling test for our optimized non-grouping GND core with two different resolution G7 and G8. What can be clearly seen in this figure is that the parallel efficiency gradually decreases as the increasing of the computing resources, which is caused by the communication overhead. Meanwhile, the strong scaling gets better as the increasing of problem size. The strong speedup of G8 reaches nearly 256 and is twice that of G7 at 1024 cores. Specially, there is a sudden drop in parallel efficiency from 4 cores to 16 cores, and which is driven by the performance bottleneck of a single computing node where all the tasks (less than 24) were assigned to. In addition, there is an inflection point on the both speedup curves when the number of processors is 32, which can be attributed to the additional network communication as the 32 cores have to be split into two computing nodes. It reveals that reducing unnecessary inter-host communication is necessary. Therefore, we adjusted the test plan so that the number of processors participating in the computing is an integer multiple of 24 (the number of CPU cores in each computing nodes).

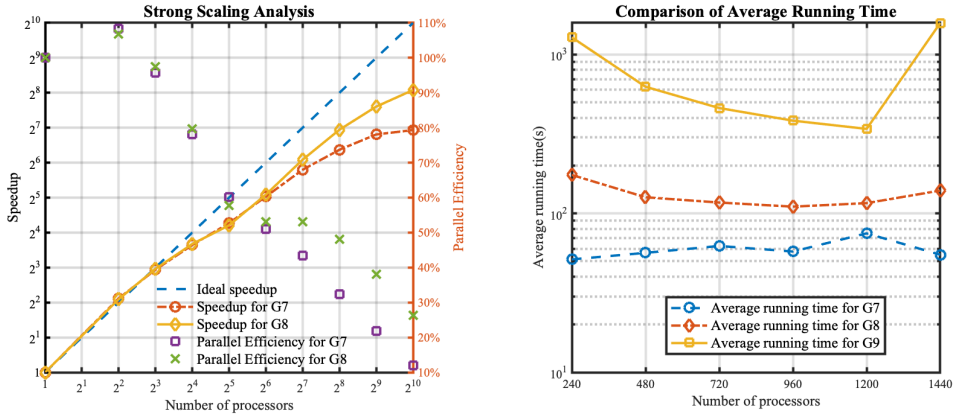


Figure 6: Strong scaling test of non-grouping GND core with G7 and G8 (left) and comparison of average running time under different process numbers with G7, G8 and G9 (right)

5.2 Average running time analysis of non-grouping GND core

In terms of the relationship between the problem size and the number of processors, we conducted the following experiments. For three kinds of problem size G7, G8 and G9, we

ran the non-grouping GND core for three times with 240, 480, 720, 960, 1200 and 1440 processors, respectively.

As shown in Tab. 3, if the resolution accuracy increases by one level, the problem size is expanded by four times. In the vertical direction, the approximate relationship between the resolution accuracy and the average running time can be seen in Fig. 6 (right). In the horizontal direction, what we expected is that the average time consumption could be decrease with the addition of the number of processors. However, the results shown in Fig. 6 (right) are slightly different. Following the addition of the number of processors, there has been a slight increase in the average time consumption for G7. Such increasing in time stems from the fact that too many processors were involved in the small problem and the extra inter-process communication increased the running time instead. For resolution G8, the average running times of 1200 and 1440 processors are little higher than those of less processors. Likewise, the average running time reached a low point of 1200 processors for resolution G9. The reason for rebound at the end of the curve stems from the communication overhead. Comparing the curves (regardless of rebound part) of G8 and G9, the slope increases obviously with scaling up the problem size, and which means that the larger the problem size, the more obvious the performance improvement with the addition of the number of processors.

5.3 Performance analysis of parallel GND core

In order to optimize I/O performance, a grouping method was proposed in Section 4, and that grouping method could introduce additional communication overhead in local groups. Therefore, we carried out a series of experiments to examine the impact of grouping method and group size on the performance.

Firstly, most experimental results show that introducing grouping method will increase the average running time. However, as shown in Fig. 7, there are still some exceptions where grouping method can significantly reduce average running time or is basically the same as non-grouping, such as running with 480, 720 and 960 processors for G7, 960 and 1200 processors for G8 and 1440 processors for G9. In particular, for running parallel GND core with 1440 cores for resolution G9, average time consumption is reduced to one third of non-grouping method as shown in Fig. 7(c). These results show that grouping method can definitely improve the performance when suitable processors number was chosen according the problem size.

In addition to the running time, the reading and writing time used to perform I/O operations are the focus in our experiments. Actually, the reading time and writing time mainly refer the execution time of function *pull_element_types_full* and *gcm_output_file*, respectively. For reading data from NC files, the host process gathered all the *read* requests from other guest processes in local communication group firstly. After got all the *read* requests, those requests were sorted and then combined into one request and that was sent to underlying PnetCDF interfaces to perform real reading operations. Finally, the host process scattered the obtained data to other guest processes according to their requests. For writing data to NC file, a similar process was performed as well.

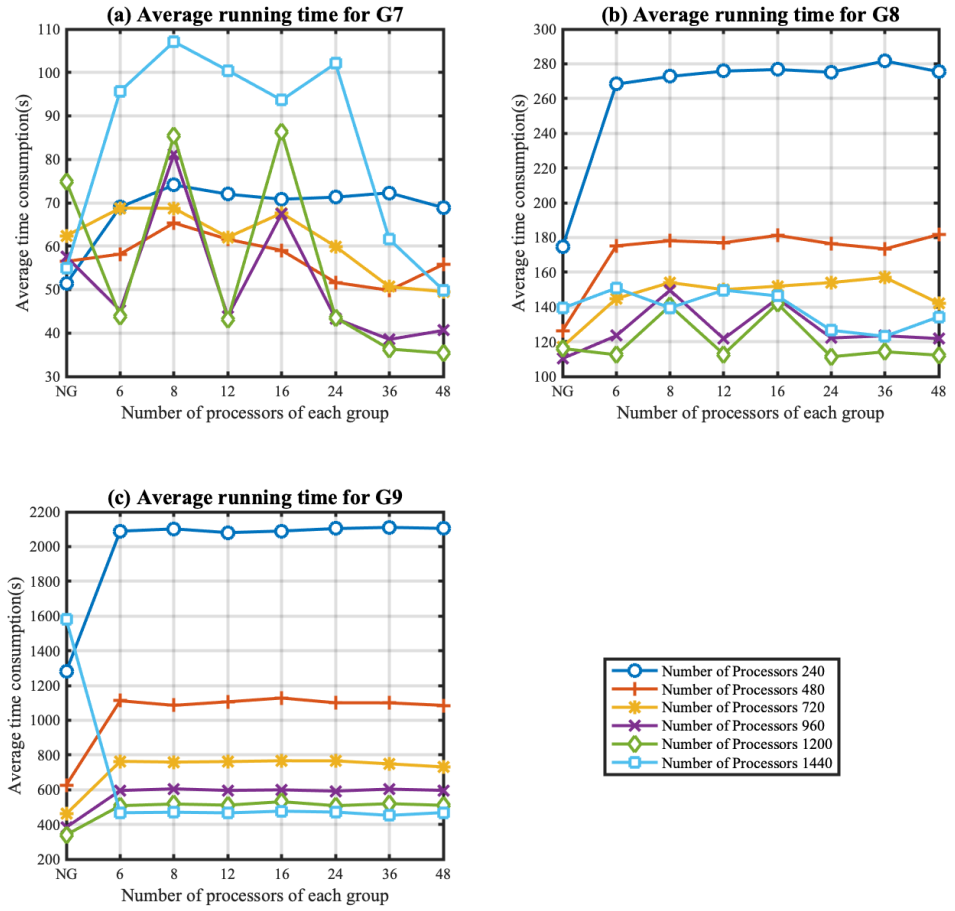


Figure 7: Comparison of average running time of parallel GND core with different number of processors for non-grouping (NG) and different number of processors of each group

Therefore, the average reading and writing time are mainly composed of execution time (T_1) of PnetCDF interfaces and communication time (T_2) of gathering and scattering. To the smaller problem size G7 and G8, the reduction of time T_1 caused by combining multiple small data requests into bigger data blocks is greater than the increase of T_2 with the increasing problem size, and so the average reading and writing time of parallel GND core is less than those of non-grouping one as shown in Figs. 8 (a)-8(d). In contrast, the results are just opposite with increasing problem size to G9 as shown in Figs. 8 (e) and 8(f). However, it is also obvious that the reading and writing time can be reduced by increasing the group size, especially for the writing time with group size 36 and 48 in Figs. 8(b), 8(d) and 8(f).

6 Related work

Generally, the process of running an application on a computing system can be abstracted as obtaining instructions and data from a storage hierarchy, processing the data by executing instructions in the CPU and writing the results back to the storage. In order to

run massive scientific applications effectively, distributed HPC architectures, shared memory parallelism (SMP) and distributed big data architectures were deployed to undertake the workload [Tavara (2019)]. In terms of performance, massive scientific applications could be divided into computing-sensitive applications and I/O-sensitive applications. Therefore, optimizing computing efficiency and improving I/O performance are two main approaches.

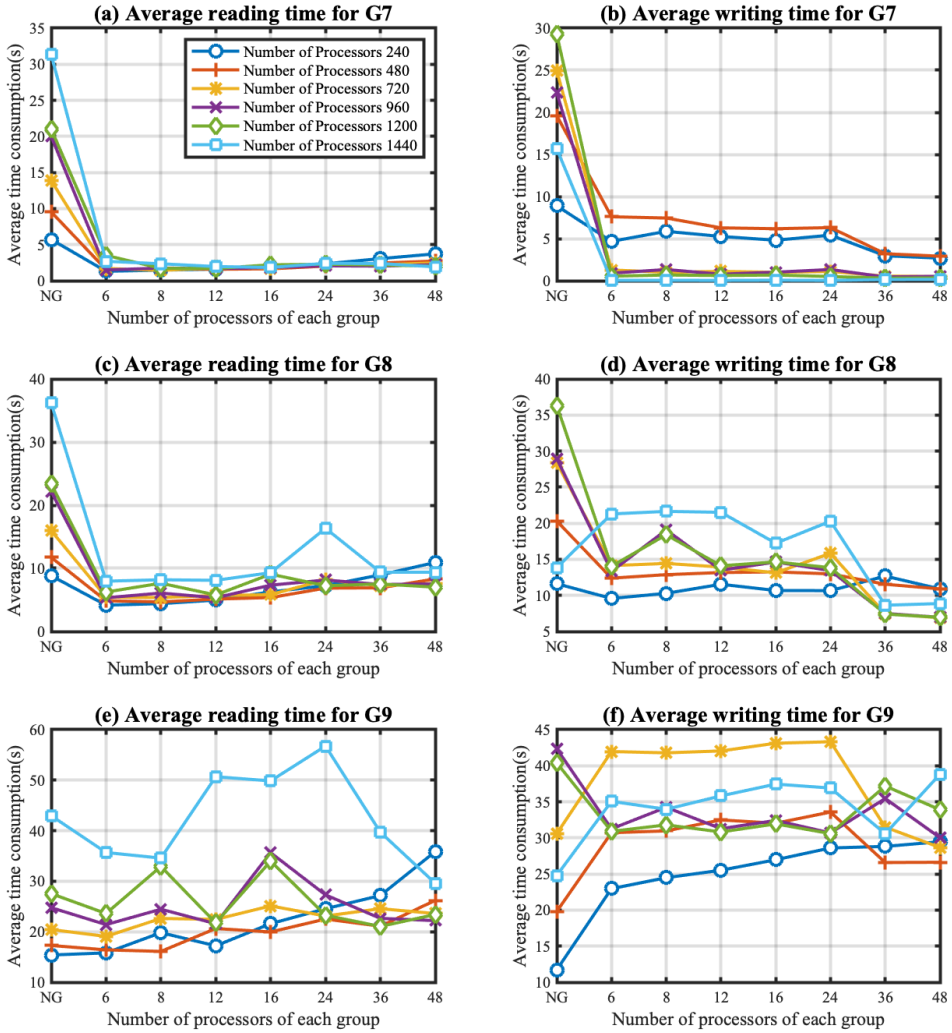


Figure 8: Comparison of average reading and writing time of parallel GND core with different number of processors and different number of processors of each group

To speed up the computing, several parallel algorithms and data structures have been developed to make full use of the underlying hardware resources to solve problems in different fields. Codreanu et al. [Codreanu, Dröge, Williams et al. (2016)] show that 10-fold speedups could be achieved by modifying the input data structure and combining memory access in the implement of support vector machine, which could match the

memory access pattern on the platform and provide more the memory throughput. Moreover, with the rapid development of GPU technology in recent years, GPU platforms have been gradually adopted to run computing-sensitive applications, thereby improving operating efficiency. He et al. [He, Bai, Ouyang et al. (2019)] proposed a parallel cloud-derived wind inversion algorithm based on GPU framework, which takes advantage of GPU cores to run each iteration and the acceleration ratio of that algorithm is up to 112.

To improve the performance of I/O intensive applications, reducing the data moving between CPUs and memory by designing new data structures and optimizing data access strategies is one of the most common methods. Sarje et al. [Sarje, Song, Jacobsen et al. (2015)] utilized reducing communication approaches to minimize the data movement both inter- and intra-nodes, as well as improving cache efficiency by predictive ordering techniques, which optimized the unstructured mesh-based MPAS-Ocean platform. In addition, coalescing multiple small non-contiguous I/O requests into fewer large contiguous ones is another popular technique, known as collective I/O or two-phase I/O [Rosario, Bordawekar and Choudhary (1993)], which is a well-known parallel I/O strategy for shared file access. In the collective I/O approach, all the I/O requests producing from the processes running simultaneously are sent to few processes that were selected as "aggregators" instead of the underlying parallel file systems, such as Lustre, PVFS, GPFS. In the communication phase, aggregators collective all the requests, coalesce them into few sorted ones. In the I/O phase, aggregators wait for the finishing of real I/O operations and send responses back to all the other processes. Kumar et al. [Kumar, Vishwanath, Carns et al. (2012)] presented an algorithm to enable a three-phase scheme, which restructures simulation data into large blocks to further improve the I/O performance. TAPIOCA [Tessier, Vishwanath and Jeannot (2017)] is an I/O library based on MPI-IO, which implements an efficient topology-aware two-phase I/O scheme and effectively reduce the idle time during the communication phase. PnetCDF [Li, Liao, Choudhary et al. (2003)] is an enhanced netCDF library, which takes advantage of the underlying MPI-IO to support efficient data storage and access as well as a new parallel interface for reading and writing netCDF dataset directly.

7 Conclusion

This study has presented the parallel implementation and optimization to a new global nonhydrostatic dynamical (GND) core running on the commercial auxiliary computing system of national supercomputing center in Wuxi, which is a supercomputing platform based on Intel x86 CPUs. The GND core is part of the prototype of a Global - Regional Integrated forecast System (GRIST), and the purpose of which is to develop a new fully-fledged atmospheric general circulation model.

Utilizing MPI technique, parallelization of the GND core was implemented by introducing boundary and halo domains, and meanwhile some sophisticated data structures and interfaces were designed to improve the performance as well as supporting modification to the size of boundary and halo domains according to the variable accuracy. Moreover, a grouping solution was proposed in this paper to improve the I/O performance and to avoid resulting SWOs when the number of MPI processes is more

than 6000. The results demonstrate our approach has a better strong scaling and improves the performance of GND core significantly.

Acknowledgement: This work was supported by the National Key Research and Development Program of China under Grant No. 2017YFC1502203.

Funding Statement: The author(s) received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Codreanu, V.; Dröge, B.; Williams, D.; Yasar, B.; Yang, P. et al. (2016): Evaluating automatically parallelized versions of the support vector machine. *Concurrency and Computation: Practice and Experience*, vol. 28, no. 7, pp. 2274-2294.

Du, Q.; Gunzburger, M. D.; Ju, L. (2003): Constrained centroidal voronoi tessellations for surfaces. *SIAM Journal on Scientific Computing*, vol. 24, no. 5, pp. 1488-1506.

Dubos, T.; Dubey, S.; Tort, M.; Mittal, R.; Meurdesoif, Y. et al. (2015): Dynamico-1.0, an icosahedral hydrostatic dynamical core designed for consistency and versatility. *Geoscientific Model Development*, vol. 8, no. 10, pp. 3131-3150.

Folk, M.; Heber, G.; Koziol, Q.; Pourmal, E.; Robinson, D. (2011): An overview of the hdf5 technology suite and its applications. *Proceedings of the EDBT/ICDT Workshop on Array Databases*, pp. 36-47.

Gao, K.; Liao, W.; Choudhary, A.; Ross, R.; Latham, R. (2009): Combining i/o operations for multiple array variables in parallel netcdf. *IEEE International Conference on Cluster Computing and Workshops*, pp. 1-10.

Girard, C.; Plante, A.; Desgagné, M.; Cowan, R. M.; Côté, J. et al. (2014): Staggered vertical discretization of the canadian environmental multiscale (gem) model using a coordinate of the log-hydrostatic-pressure type. *Monthly Weather Review*, vol. 142, no. 3, pp. 1183-1196.

He, L.; Bai, H.; Ouyang, D.; Wang, C.; Wang, C. et al. (2019): Satellite cloud-derived wind inversion algorithm using GPU. *Computers, Materials & Continua*, vol. 60, no. 2, pp. 599-613.

Kumar, S.; Vishwanath, V.; Carns, P.; Levine, J. A.; Latham, R. et al. (2012): Efficient data restructuring and aggregation for i/o acceleration in pidx. *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1-11.

LaSalle, D.; Patwary, M. M. A.; Satish, N.; Sundaram, N.; Dubey, P. et al. (2015): Improving graph partitioning for modern graphs and architectures. *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pp. 1-4.

Li, J.; Liao, W. K.; Choudhary, A.; Ross, R.; Thakur, R. et al. (2003): Parallel netcdf: a high-performance scientific i/o interface. *SC' 03: Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 39-39.

Ringler, T.; Ju, L.; Gunzburger, M. (2008): A multiresolution method for climate system modeling: application of spherical centroidal voronoi tessellations. *Ocean Dynamics*, vol. 58, no. 5, pp. 475-498.
Rosario, J. M. D.; Bordawekar, R.; Choudhary, A. (1993): Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Computer Architecture News*, vol. 21, no. 5, pp. 31-38.
Sarje, A.; Song, S.; Jacobsen, D.; Huck, K.; Hollingsworth, J. et al. (2015): Parallel performance optimizations on unstructured mesh-based simulations. *Procedia Computer Science*, vol. 51, no. C, pp. 2016-2025.
Smolarkiewicz, P. K.; Kühnlein, C.; Grabowski, W. W. (2017): A finite-volume module for cloud-resolving simulations of global atmospheric flows. *Journal of Computational Physics*, vol. 341, pp. 208-229.
Tavara, S. (2019): Parallel computing of support vector machines: a survey. *ACM Computing Surveys*, vol. 51, no. 6, pp. 1-38.
Tessier, F.; Vishwanath, V.; Jeannot, E. (2017): Tapioca: an i/o library for optimized topology-aware data aggregation on large-scale supercomputers. *IEEE International Conference on Cluster Computing*, pp. 70-80.
Yu, R.; Zhang, Y.; Wang, J.; Li, J.; Chen, H. et al. (2019): Recent progress in numerical atmospheric modeling in china. *Advances in Atmospheric Sciences*, vol. 36, no. 9, pp. 938-960.
Zängl, G.; Reinert, D.; Rípodas, P.; Baldauf, M. (2015): The icon (icosahedral non-hydrostatic) modelling framework of dwd and mpi-m: description of the non-hydrostatic dynamical core. *Quarterly Journal of the Royal Meteorological Society*, vol. 141, no. 687, pp. 563-579.
Zhang, Y. (2018): Extending high-order flux operators on spherical icosahedral grids and their applications in the framework of a shallow water model. *Journal of Advances in Modeling Earth Systems*, vol. 10, no. 1, pp. 145-164.
Zhang, Y.; Li, J.; Yu, R.; Zhang, S.; Liu, Z. et al. (2019): A layer-averaged nonhydrostatic dynamical framework on an unstructured mesh for global and regional atmospheric modeling: model description, baseline evaluation, and sensitivity exploration. *Journal of Advances in Modeling Earth Systems*, vol. 11, no. 6, pp. 1685-1714.