A Peta-scalable CPU-GPU Algorithm for Global Atmospheric Simulations

Chao Yang[1,5], Wei Xue[2,3], Haohuan Fu[3], Lin Gan[2,3], Linfeng Li[2], Yangtong Xu[2,3], Yutong Lu[4], Jiachang Sun[1], Guangwen Yang[2,3] and Weimin Zheng[2,3]

1. Institute of Software, Chinese Academy of Sciences, Beijing, China
2. Department of Computer Science & Technology, Tsinghua University, Beijing, China
3. Ministry of Education Key Laboratory for Earth System Modeling, and Center for Earth System Science, Tsinghua University, Beijing, China
4. Department of Computer Science & Technology, National University of Defense Technology, Changsha, Hunan, China
5. State Key Laboratory of Space Weather, Chinese Academy of Sciences, Beijing, China

yangchao@iscas.ac.cn, {xuewei,haohuan}@tsinghua.edu.cn, {lin.gan27,linfeng.li1986,xuyangtong}@gmail.com, luyut@sina.com, sun@mail.rdcps.ac.cn, {ygw,zwm-dcs}@tsinghua.edu.cn

Abstract

Developing highly scalable algorithms for global atmospheric modeling is becoming increasingly important as scientists inquire to understand behaviors of the global atmosphere at extreme scales. Nowadays, heterogeneous architecture based on both processors and accelerators is becoming an important solution for large-scale computing. However, large-scale simulation of the global atmosphere brings a severe challenge to the development of highly scalable algorithms that fit well into state-of-the-art heterogeneous systems. Although successes have been made on GPU-accelerated computing in some top-level applications, studies on fully exploiting heterogeneous architectures in global atmospheric modeling are still very less to be seen, due in large part to both the computational difficulties of the mathematical models and the requirement of high accuracy for long term simulations.

In this paper, we propose a peta-scalable hybrid algorithm that is successfully applied in a cubed-sphere shallow-water model for global atmospheric simulations. We employ an adjustable partition between CPUs and GPUs to achieve a balanced utilization of the entire hybrid system, and present a pipe-flow scheme to conduct conflict-free inter-node communication on the cubed-sphere geometry and to maximize communication-computation overlap. Systematic optimizations for multithreading on both GPU and CPU sides are performed to enhance computing throughput and improve memory efficiency. Our experiments demonstrate nearly ideal strong and weak scalabilities on up to 3,750 nodes of the Tianhe-1A. The largest run sustains a performance of 0.8 Pflops in double precision (32% of the peak performance), using 45,000 CPU cores and 3,750 GPUs.

Categories and Subject Descriptors D.1.3 [*Programing Techniques*]: Concurrent programming; J.2 [*Physical Sciences and Engineering*]: Earth and atmospheric sciences; F.2.1 [*Analysis of Algorithms and Problem Complexity*]: Numerical Algorithms and Problems

Keywords parallel algorithm; atmospheric modeling; GPU; heterogeneous system; communication-computation overlap; scalability

PPoPP'13, February 23–27, 2013, Shenzhen, China.

1. Introduction

Numerical simulation of the global atmosphere, as a key component in climate modeling, is one of the most computationally challenging problems in scientific computing. As scientists inquire to understand dynamic behaviors of the global atmosphere at increasingly fine resolutions [9, 16, 19, 27], the development of highly scalable algorithms for global atmospheric modeling is becoming an urgent demand. Scalable atmospheric solvers not only enable high-fidelity simulation of realistic problems but also lead to dramatic reduction in time-to-solution and substantial increase in accuracy.

Nowadays, heterogeneous architecture based on both CPUs and GPUs is becoming an important solution for large-scale computing. Successes have been made in applying efficient hybrid algorithms to some top-level applications, such as N-body simulations [7, 8, 11], biofluidics simulations [3] and phase-field simulations [26]. Although some promising approaches have been proposed to take advantage of GPU accelerations in regional weather predictions (e.g., [10, 14, 24, 25]), studies on fully exploiting heterogeneous architectures in global atmospheric modeling are still undergoing.

There are several difficulties in efficiently running a global atmospheric model on a petascale heterogeneous supercomputer. One comes from the reality that the global atmosphere is defined on a large computational area (i.e., the Earth) and exhibits a broad range of different spatial and temporal scales. These characteristics of the atmosphere require a global or stencil-based method instead of a particle-based method which is favorable on reducing the coupling of the system (e.g., [8]).

As a success in GPU-accelerated stencil computing, a petaflop performance in single precision has been achieved by Shimokawabe et al. [26] for phase-field simulations on the TSUBAME 2.0 supercomputer (2011 Gordon Bell Prize); but the double-precision performance (260 Tflops) in the same work, which is essential for atmospheric modeling, is relatively low. In order to conduct large-scale global atmospheric modeling on modern heterogeneous systems, we propose a scalable algorithm that works well for a

global shallow-water model and sustains promising petascale performance in double precision. In the new algorithm, we apply an adjustable partition between CPUs and GPUs to achieve a balanced utilization of the entire system. And based on that, systematic optimizations for multithreading on both GPU and CPU sides are performed to enhance double-precision computing throughput as well as to improve memory efficiency.

Another difficulty in petascale simulation of the global atmosphere is the selection of the computational mesh for the Earth. The traditional latitude-longitude (Lat-Lon) mesh has been serving the atmospheric community for several decades. However, as the resolution becomes finer, the Lat-Lon mesh becomes unable to maintain satisfying load-balance due to the non-uniformity. For example, it has been claimed by Putman [18] that the Lat-Lon mesh may only scales to a few thousand processors at 0.1° horizontal resolution. In this study we use the cubed-sphere mesh among several choices [13, 22, 23, 28] primarily because: (1) it provides a good load-balance even when the number of processors is substantially large; and (2) each patch of the partition helps improve the overall performance in aligned memory access.

Due to the intrinsic curvilinear nature of the sphere, message passing pattern on the cubed-sphere is complicated compared to structured stencil computations such as the work of Shimokawabe et al. [26] in which the computational domain is a three-dimensional cube. To that end, we propose a "pipe-flow" communication scheme for the rearrangement of send/receive pairs across sub-block boundaries in order to maximize communication-computation overlap for the cubed-sphere geometry and to dramatically reduce communication overhead.

The rest of this paper is organized as follows. In Section 2, we introduce the mathematical model and numerical methods used in our global atmospheric simulation. Major algorithms proposed in this paper are presented in Section 3 in detail, after which some key implementation and optimization strategies on the Tianhe-1A are given in Section 4. We then show by several large-scale numerical tests in Section 5 that the proposed hybrid algorithm is petascalable on the Tianhe-1A. The paper is concluded in Section 6.

2. Equations and Discretizations

Among several equation sets that can be used to model the global atmosphere, shallow-water equations (SWEs) exhibit most of the essential characteristics of the atmosphere, thus can be used as a test bed for the development of new algorithms. The SWEs on a rotating sphere can be written as a system of conservation laws:

$$\begin{cases} \dfrac{\partial h}{\partial t} + \nabla \cdot (h\mathbf{v}) = 0, \\ \dfrac{\partial (h\mathbf{v})}{\partial t} + \nabla \cdot (h\mathbf{v} \otimes \mathbf{v}) = \Psi_C + \Psi_G, \end{cases} \tag{1}$$

where h is the thickness of the atmosphere, $\mathbf{v}$ is the velocity vector defined on the surface of the sphere. The two source terms $\Psi_C = -fh(\hat{\mathbf{k}} \times \mathbf{v})$ and $\Psi_G = -gh\nabla(h+b)$ are due to the Coriolis force and the gravity force, respectively. Here $\hat{\mathbf{k}}$ is the unit outward normal on the sphere, f is a Coriolis parameter, g is the gravitational constant and b is the height of the spherical surface describing a variable bottom topography (e.g., mountains).

In this study, we employ a cubed-sphere mesh that is defined by mapping an inscribed cube of the sphere to the surface, as shown in Fig. 1. The computational domain is the six faces of the cube, corresponding to the six patches on the sphere. An advantage of the cubed-sphere geometry is that the SWEs, when written in local coordinates, have an identical expression on the six patches; that is

$$\frac{\partial Q}{\partial t} + \frac{1}{\Lambda}\frac{\partial (\Lambda F^1)}{\partial x^1} + \frac{1}{\Lambda}\frac{\partial (\Lambda F^1)}{\partial x^2} + S = 0, \tag{2}$$

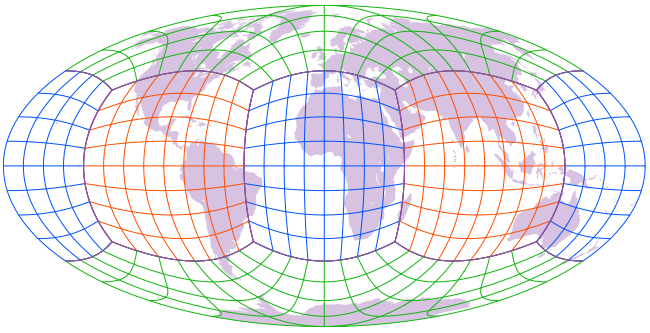

Figure 1. A cubed-sphere mesh on the sphere. The cubed-sphere is obtained by mapping an inscribed cube of the sphere to the surface. Mesh lines on the cubed-sphere coincide with great circles.

where $(x^1, x^2) \in [-\pi/4, \pi/4]$ are the local coordinates of a patch, $Q = (h, hu^1, hu^2)^T$ is the prognostic variable, $F^i = u^i Q$ $(i = 1, 2)$ are the convective fluxes and $S = (0, S_1, S_2)^T$ is the source term. Note that the two contravariant components of the velocity, u^1 and u^2, are non-orthogonal. The source term becomes more complicated due to the non-orthogonality of the cubed-sphere, e.g.,

$$S_1 = -\frac{f}{\Lambda}\sum_{i=1}^{2}\left(g_{2i}hu^i\right) + gh\sum_{i=1}^{2}\left(g^{1i}\frac{\partial Z}{\partial x^i}\right) + \sum_{i,j=1}^{2}\Gamma^1_{ij}(hu^i u^j),$$

where $Z = h + b$ is the surface level of the atmosphere. Variable coefficients found therein such as Λ, Γ^k_{ij}, g_{ij} and g^{ij} all have fixed expressions that only depend on their geometric positions; details can be found in, e.g., [20, 21].

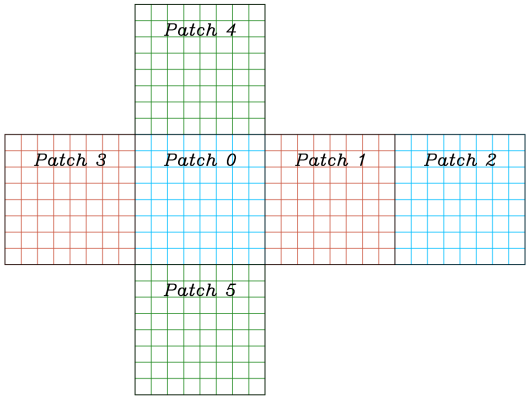

Figure 2. The computational domain of a cubed-sphere consists of six patches that are mapped from the six faces of a cube. Each patch is covered by a uniform rectangular mesh.

Suppose the cubed-sphere is equal-distantly meshed in the computational domain, with $N \times N$ mesh cells in each patch, as seen in Fig. 2. Then we may spatially discretize the SWEs (2) by using, for instance, a cell-centered finite volume method. All prognostic variables h, hu^1 and hu^2 are simultaneously approximated within each mesh cell as a single vector variable

$$Q^k_{ij}(t) = \frac{1}{\Lambda^k_{i,j}}\int_{C(i,j,k)} Q(t)d\sigma, \quad \Lambda^k_{i,j} = \int_{C(i,j,k)} d\sigma,$$

where $C(i,j,k)$ is a mesh cell with index (i,j) in patch k and $d\sigma = \Lambda dx^1 dx^2$. Then a solution vector is composed as $X(t) = [Q^k_{ij}(t)]$ at time frame t. Integrating (2) over each mesh cell leads

to a semi-discrete system:

$$\frac{\partial X(t)}{\partial t} + \mathcal{L}(X(t)) = 0, \tag{3}$$

where $\mathcal{L}(X(t)) = [L_{ij}^k(t)]$ and

$$L_{ij}^k(t) = \frac{4N}{\Lambda_{ij}^k \pi} \int_{\partial C_{i,j,k}} (F^1(t), F^2(t)) \cdot \mathbf{n} ds + S_{ij}^k(t), \tag{4}$$

with $\mathbf{n}$ being the outward unit normal.

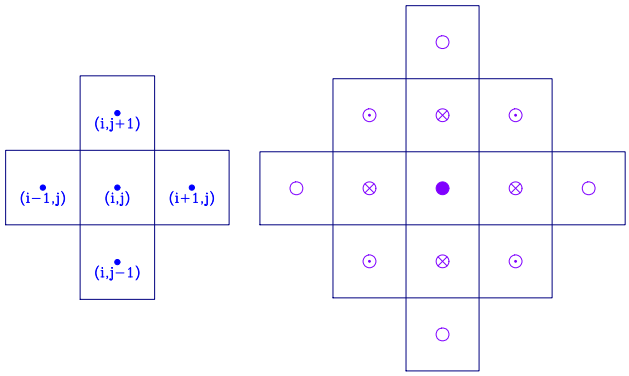

Figure 3. Left: State reconstruction in cell (i, j), values in the adjacent four cells are needed. Right: The 13-point stencil exhibits a diamond shape. Each dot represents a mesh cell on which the three prognostic variables are evaluated.

Cell edge integrations of F^i in (4) are done in a two-step manner. The first step is to reconstruct Q on each cell edge from both limit sides using the value of Q_{ij}^k on several adjacent neighbors. For example, as shown in the left panel of Fig. 3, two reconstructed states of q in the x^1 direction are obtained via

$$q^+_{i-1/2,j} = \tfrac{16}{24} q_{i,j} + \tfrac{1}{24}(q_{i,j-1} + q_{i,j+1} + 3q_{i-1,j} - q_{i+1,j}),$$
$$q^-_{i+1/2,j} = \tfrac{16}{24} q_{i,j} + \tfrac{1}{24}(q_{i,j-1} + q_{i,j+1} + 3q_{i+1,j} - q_{i-1,j}).$$

The second step is to calculate F^i on any cell edge using a modified Osher's Riemann solver ([17]) as

$$\int_{\partial C_{i,j,k}} (F^1, F^2) \cdot \mathbf{n} ds \approx \left(\int_{\partial C_{i,j,k}} ds \right) (F^1, F^2)\big|_{Q^*} \cdot \mathbf{n}, \tag{5}$$

where Q^* is calculated from a nonlinear combination [31] of Q^- and Q^+ on the same edge. Putting the two steps together, the stencil used in the calculation of (4) exhibits a diamond shape, with 13 points in total, as shown in the right panel in Fig. 3. Due to the hyperbolic nature of the SWEs, the computation of Q^* could not be done without a proper upwinding mechanism; that is, "if-else" statements are used in the code to calculate stencils. This type of dependency complicates the problem studied here from those in other applications (e.g., [26]).

To properly pass information between neighboring patches, a two-point linear interpolation is used to calculate corrected values on halos (i.e., ghost cells) across patch interfaces. Velocity components are transformed into a same coordinates system on each patch interface in the calculation of the numerical fluxes to maintain mass conservation, which is important for long-term integrations in climate modeling.

We integrate the SWEs using a second-order accurate total variation-diminishing Runge-Kutta method [6] that reads

$$\begin{aligned} \overline{X}(t^{(m)}) &= X(t^{(m-1)}) - \Delta t \mathcal{L}(X(t^{(m-1)})), \\ X(t^{(m)}) &= \frac{1}{2}\left\{ X(t^{(m-1)}) + \overline{X}(t^{(m)}) \right\} - \frac{1}{2}\Delta t \mathcal{L}(\overline{X}(t^{(m)})), \end{aligned} \tag{6}$$

in which there are two stencil evaluations at each time step.

3. Algorithms

Before describing the algorithms, we first decompose each patch of the cubed-sphere into small sub-blocks along both dimensions in a same way. Each sub-block is managed by an MPI process that corresponds to a computing node in a CPU-GPU cluster. Halo information from neighboring sub-blocks is updated before computing the stencils in each sub-block, as shown in Fig. 4.

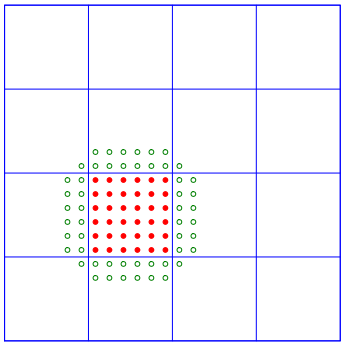

Figure 4. The halo updating pattern for a sub-block that is obtained by decomposing a patch of the cubed-sphere. Mesh cells in a sub-block are shown as solid dots and halo cells required by the sub-block are shown as empty dots.

3.1 CPU-only algorithm

In the CPU-only algorithm, the two stencil operations in (6) are carried out in a same procedure, which is described in Algorithm 1. MPI processes are utilized to manage all sub-blocks in the algorithm. In addition to that, by employing multi-threading techniques such as OpenMP, another level of parallelism can be added within each sub-block in order to further exploit the multi-core CPUs in each computing node.

Algorithm 1 CPU-only algorithm for each stencil cycle.

```
for all six patches do
    for all sub-blocks in each patch do
        Update halo information
        Interpolation on halos when necessary
        for all mesh cells in each sub-block do
            Compute stencil for the h component
            Compute stencil for the hu^1 component
            Compute stencil for the hu^2 component
        end for
    end for
end for
```

The workflow of Algorithm 1 for each stencil cycle is shown in Fig. 5, which consists of four stages: ① halo updating, ② data copying, ③ halo interpolating, and ④ stencil computing. Based on the framework of PETSc (Portable Extensible Toolkit for Scientific computation [2]), we make use of a pair of neighboring communication functions (`VecScatterBegin/End`) to update halo information. Right after that, an extra data copy is required to fill the buffer for later use. Then a linear interpolation is carried out on the halos across patch interfaces to prepare proper ghost-cell information for stencil computations. Since halo updating needs to be done before the stencil computation part, the communication can not be hidden and will eventually degrade the scalability when the number of MPI

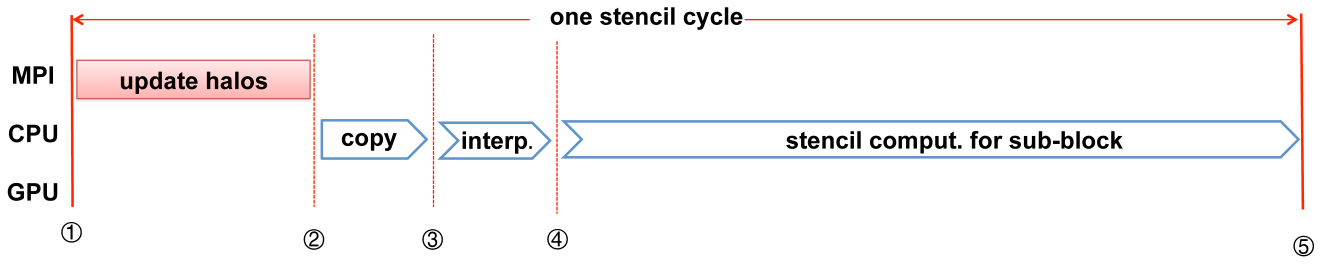

Figure 5. A CPU-only algorithm for the stencil computations. Halo updates are performed with MPI. An extra data copy is needed to prepare data for this stencil cycle before the stencil computations.

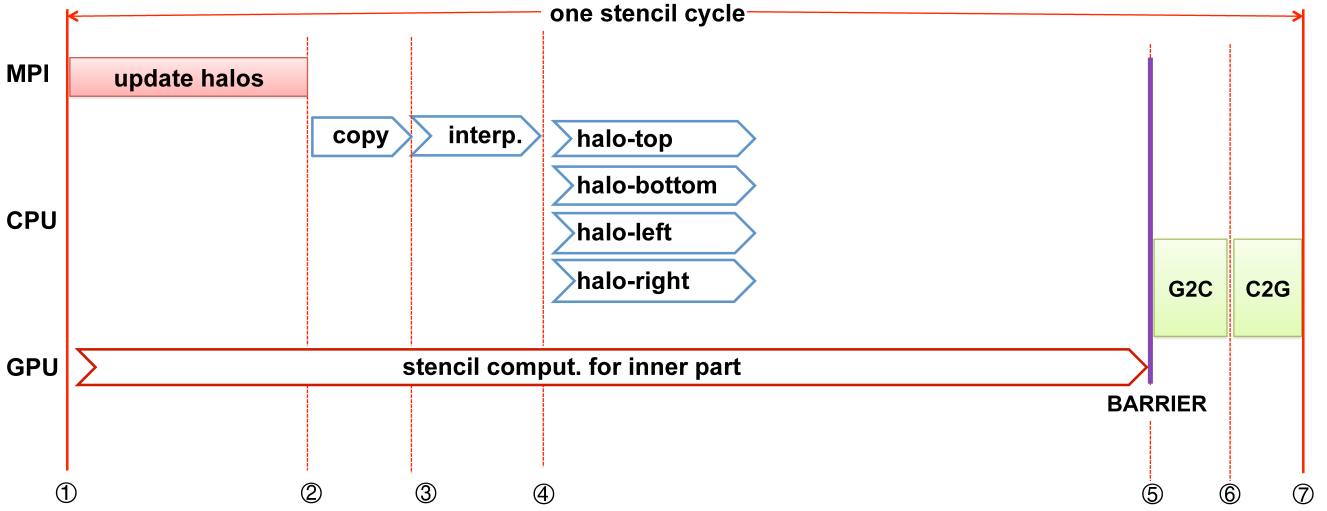

Figure 6. A hybrid CPU-GPU algorithm for the stencil computations. In the figure, "interp." refers to the interpolation on halos; "halo-top/bottom/left/right" refer to the stencil computations for the four halo areas in the outer part of the sub-block; "G2C" refers to the data movement from GPU to CPU and "C2G" refers to the data movement from CPU to GPU.

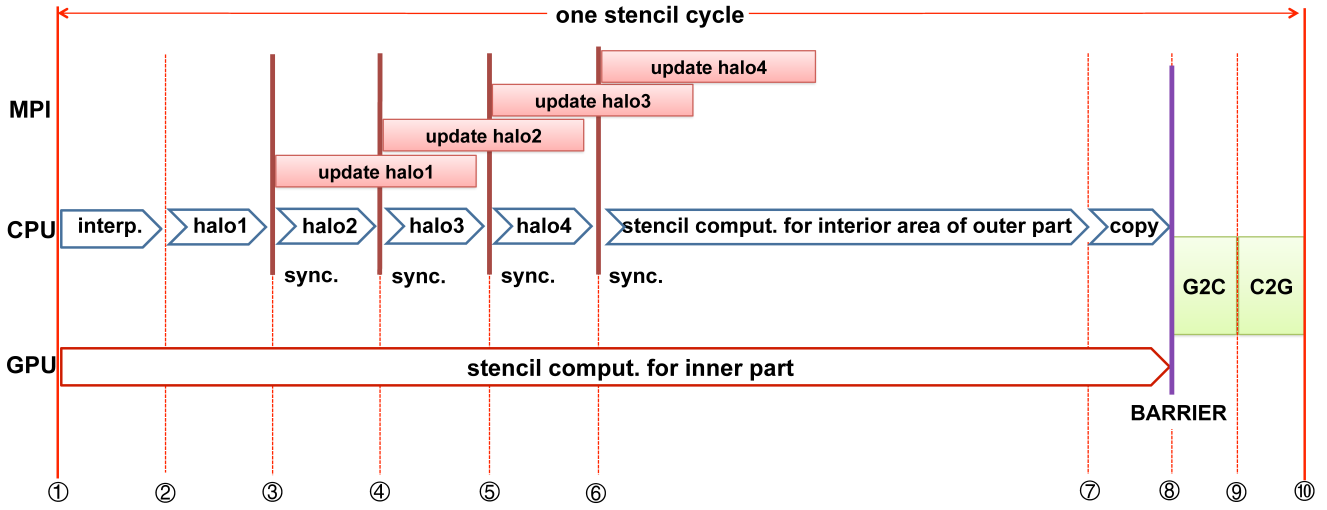

Figure 7. An optimized hybrid CPU-GPU algorithm for the stencil computations. The partition between CPUs and GPUs is adjusted to balance different computing resources and communication-computation overlap is applied in the optimized algorithm. In the figure, "halo-1/2/3/4" refer to stencil computations inside the four halo areas of the outer part.

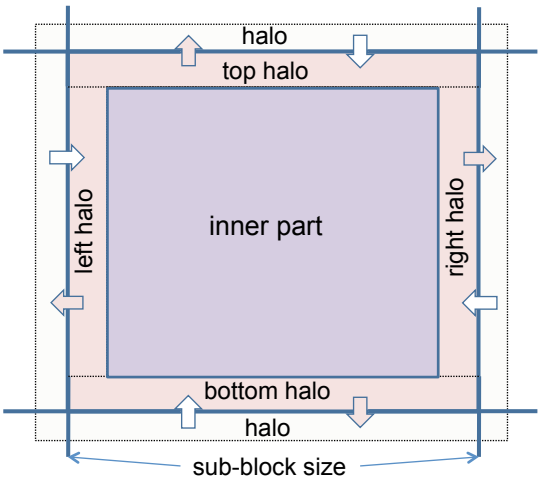

Figure 8. Each sub-block can be decomposed into an inner part that does not need halo information and a 2-layer outer part that is close to the four boundaries of the sub-block.

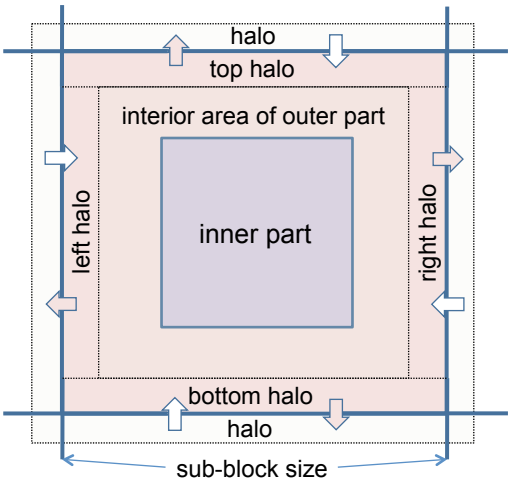

Figure 9. An adjustable partitioning of a sub-block between the CPUs and the GPUs. Compared to the original two-layer outer part, an interior area is introduced in the extended outer part. Accordingly, the size of the inner part assigned to the GPU becomes smaller.

processes becomes large. Besides, the CPU-only approach does not benefit from the GPU resources in a hybrid CPU-GPU system.

3.2 Hybrid CPU-GPU algorithm

In this subsection, we present a hybrid algorithm that is similar to the one employed by Shimokawabe et al. [26] in phase-field simulations. After decomposing the whole domain into sub-blocks, we separate each sub-block into an inner part that does not require halo information and an outer part that contains two layers of halos needed by neighboring sub-blocks, as seen in Fig. 8. Then for each computing node, we assign the GPUs to process the inner part and the CPUs to the outer part. To efficiently process the outer part with multi-threading, we divide it into four areas spanned along the four boundaries of the sub-block, as shown in the same figure. The values computed in the four areas of the outer part are halos needed by neighboring sub-blocks.

A workflow of the hybrid CPU-GPU algorithm for each stencil cycle is given in Fig 6. In the hybrid approach, the CPU code and the GPU code are executed concurrently in two OpenMP sections. While the inner part is processed by the GPUs, the CPUs perform: ① halo updating, ② data copying, ③ halo interpolating, and ④ stencil computing in the outer part. At ⑤, when the calculations on both sides are finished, data exchanging across the interior boundaries between CPUs and GPUs are carried out. The stencil computations for the outer part can be parallelized by utilizing OpenMP threads according to the four divided areas along the boundaries.

3.3 Adjustable partition between CPUs and GPUs

As shown in Fig. 8, when the GPUs are handling the inner part, which is a very large portion of the sub-block, the CPUs process a 2-layer outer part that only contains a very small portion of the sub-block (less than 1% for a 1,024×1,024 mesh). To make better use of the CPU resources and to improve the overall computing efficiency, we further propose an adjustable partition between CPUs and GPUs.

In the new algorithm, we increase the number of mesh layers in the outer part assigned to the CPUs, and decrease the size of the inner part correspondingly. In addition to the four areas that the original outer part owns, there is an extra interior area in the extended outer part, as shown in Fig. 9. This area is now processed by the CPUs instead of the GPUs to better balance the computational loads on the two sides. There are several advantages to divide the outer part into an interior area and four two-layer halo areas. Firstly, after stencil computations inside a halo area are finished, the newly calculated values in this halo area can be sent to the neighboring sub-block, which can be done in parallel with stencil computations inside the next halo area. Secondly, stencil computations in the interior area can be performed in parallel with any unfinished halo updating, right after stencil computations in the four halo areas are done. Thirdly, the code to compute stencils for the interior area is more efficient since no judgments are needed to deal with the sub-block boundaries.

Fig. 7 illustrates the workflow of the optimized hybrid algorithm for each stencil cycle. In the new algorithm, because communications and computations are overlapped, proper halo information is already available at the beginning of the stencil cycle. So while the GPUs are processing the inner part, the CPUs: ① do halo interpolation (if necessary); ② conduct stencil computations for halo area-1; ③ conduct stencil computations for halo area-2 and send the computed results to the neighbor sub-block at the same time (similar for ④ and ⑤); ⑥ perform stencil computations in the interior of the outer part after finishing computing halo area-4; ⑦ copy necessary data to the buffer. At ⑧, when the calculations on both sides are finished, data are exchanged between GPUs and CPUs by the end of the stencil cycle.

In the new algorithm, all communications for updating halos are overlapped by the computations on CPU side, which leads to good scalability as well as efficient use of the CPU computing capacity. In addition to that, we can dynamically adjust the partition of a sub-block to achieve a balanced utilization of both CPUs and GPUs. The optimal ratio to partition the sub-blocks is searched automatically by analyzing the critical path in order to minimize the elapsed time.

3.4 "Pipe-flow" scheme for the cubed-sphere

The six patches of the cubed-sphere have different communication patterns that require a carefully designed communication strategy to conduct a conflict-free message passing. For example, as seen in Fig. 2, we should avoid concurrent sending of data from patches 0-3 to patch 4; otherwise both the number of message and the volume

of data received by patch 4 are much larger other patches, which results in an imbalanced usage of the network.

Therefore, it is important to specify an optimized sequence of communications for the cubed-sphere. For that purpose, we propose a new "pipe-flow" scheme as shown in Fig. 10. In the "pipe-

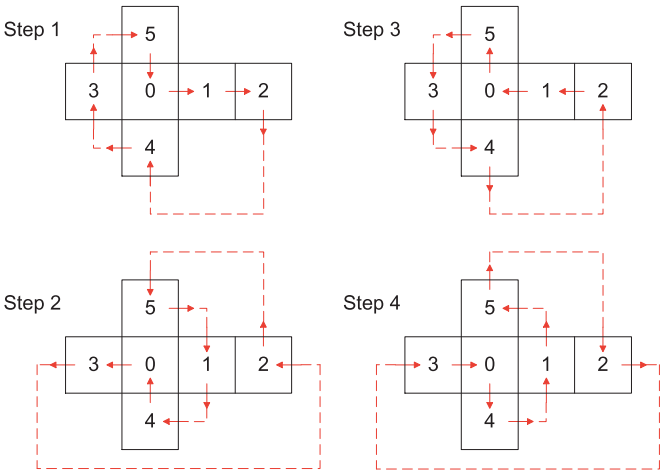

Figure 10. A "pipe-flow scheme to handle the complicated communication pattern of the cubed-sphere. There are four different steps in the arrangement of communications, shown as four panels in the figure. The arrows indicate the directions that data enter and exit the six patches like a flow.

flow" scheme, there are four different steps in the arrangement of communications. Communications are done like a flow going along a closed loop covering the six patches of the cubed-sphere, with inlet/outlet directions of the flow on each patch showing in the figure.

The direction of the "pipe-flow" inside each patch is decided by the inlet and outlet directions. The flow goes straight if the inlet and outlet of the flow are on opposite sides of the patch. Otherwise, directions of the flow are more complicated, as shown in Fig. 11.

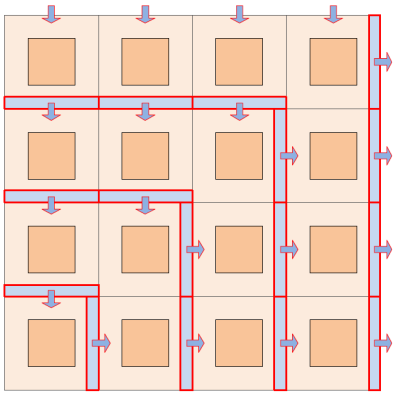

Figure 11. Directions of the "pipe-flow" inside a specific patch of the cubed-sphere. The square shaded regions represent the inner parts of sub-blocks, and the narrow shade regions represent the halos to be sent.

At any step of the "pipe-flow" scheme, each process only has one "send" and one "receive" to communicate with other processes. In this way, balanced and efficient message exchange steps are achieved for inter-process communication, leading to substantial improvement of the parallel performance when the number of processes is large.

4. Implementation & Optimization on Tianhe-1A

4.1 The Tianhe-1A supercomputer

As a petascale supercomputer, Tianhe-1A [32] features an MPP architecture of hybrid CPU-GPU computing. Unlike GPU-rich petascale systems such as the TSUBAME 2.0 [26], the deployment of CPUs and GPUs is different in the design of the Tianhe-1A supercomputer. In the system, there are totally 7,168 computing nodes, each of which consists of two six-core Intel X5670 CPUs with 32GB local memory and one NVIDIA M2050 GPU with 3GB onboard memory. The peak performance of the whole system is 4.7 Pflops in which the 100,352 GPU cores (i.e., 3,211,264 CUDA cores) provide around 3.7 Pflops and the other 1.0 Pflops are provided by the 86,016 CPU cores.

A proprietary high-speed interconnection network, the TH-net [30], is designed and implemented to enhance the communication capabilities of the system. The topology of the TH-net is an opto-electronic hybrid, hierarchical fat tree. The MPI implementation on the Tianhe-1A is customized to achieve high-bandwidth and low-latency data transfers, and the inter-node bandwidth of point-to-point unidirectional MPI operations is as high as 6,340 MB/s.

4.2 Optimization of the GPU stencil code

In the Tianhe-1A, each node is equipped with an NVIDIA Fermi M2050 GPU consisting of 14 streaming multiprocessors (SM) and 3 GB onboard memory. Each SM contains 32 streaming processors that can compute concurrently in a similar flavor to vector processors. Meanwhile, each SM also provides 64 KB local storage that can be configured as either 48-KB shared memory plus 16-KB L1 cache (the default mode) or 16-KB shared memory plus 48-KB L1 cache. To exploit the computing capacity of the GPUs in the Tianhe-1A, we employ CUDA version 4.0 for GPU programming.

According to (6), for each time step, there are two stages of stencil calculations that can be performed in a similar way. The only differences are the input/output vectors and their coefficients. The output vector of the first stencil calculation is used as the input vector of the second stencil calculation. Therefore, instead of copying the result back to the host (CPU) memory after the first stage is done, the data can remain in the GPU memory for the later use in the second stage. Only the information across interior boundaries needs to be exchanged with the host in between of the two stages. For that purpose, we map the two stages of stencil calculations in (6) into two CUDA kernel functions on the GPU in order to reduce the cost of data transfer. In addition to that, we implement another two kernels to reorder and transpose the interior boundary data that are needed by the CPUs.

On the GPU side, the two stencil kernels are clearly the most time-consuming part. A systematic optimization is performed on both the computation and memory operations to maximize the throughput for the stencil kernels. To provide a detailed description about different optimization techniques that we have considered, we go through a number of different versions of the CUDA codes in what follows.

4.2.1 Baseline

As a starting point, a baseline GPU code can be directly implemented by parallelizing the stencil computations inside the inner part of the sub-block. As shown in Fig. 12, for a NX by NY sub-block, we employ a thread grid that contains BX by BY thread blocks. Each thread block consists of TX by TY different threads. The configuration of the grid and block should satisfy that TX·BX=NX and TY·BY=NY. In this way, each point in the mesh is processed with a GPU thread. Note that for the GPU side, we do not need to compute the stencils for the mesh cells that are in the halo (shown as empty dots in Fig. 12). Therefore we can store all mesh

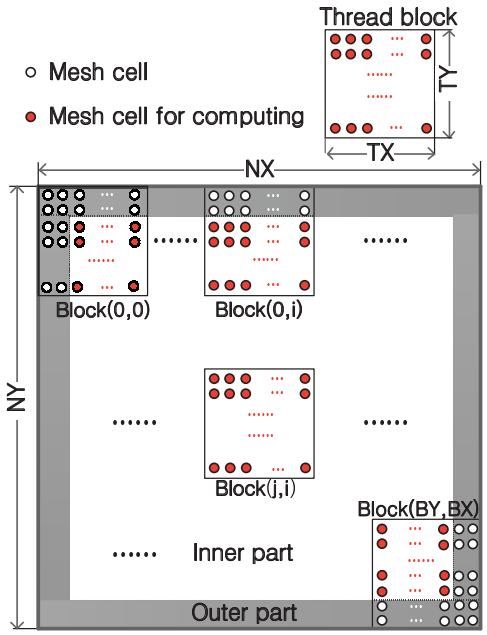

Figure 12. Strategy to compute stencils for the inner part of a sub-block using concurrent GPU threads. Mesh cells in the inner part are drawn as solid dots. And mesh cells in the halo which are needed by the inner part are drawn as empty dots.

cells of a sub-block in the GPU memory but only perform stencil calculations for the mesh cells owned by the inner part (shown as solid dots in Fig. 12).

Choosing the right size for the thread block is important for achieving a high parallel performance on the GPU platform. A general rule is to keep both TX and TY as multiples of 16. This is mainly because, on the NVIDIA M2050 GPU, threads are scheduled and executed in the unit of half-warp (16 threads). Therefore, by grouping and aligning both the arithmetic operations and memory accesses in the multiples of 16, we can generally achieve a more efficient utilization of both the 32 cores in each SM and the 16 memory banks of the shared memory.

For our specific kernels of computing nonlinear stencil terms, our experiments show that 16 by 16 is a proper thread block configuration (one thread per each point) that provides a good balance among different resources. As shown in the first point of Fig. 13, our baseline code processes a 1,024×1,024 sub-block in 117 ms.

4.2.2 Computing auxiliary vectors in run time

Several variable coefficients such as the tensor terms, the Coriolis source term and the topographic term, are needed in the evaluation of the nonlinear stencils. Note that those coefficients are only dependent of their geometry positions and remain unchanged during the whole calculation. Therefore it is a standard practice to compute and store them as auxiliary vectors for reuse. However, on the GPU side, the kernel functions need to read in up to 20 different auxiliary vectors, which is a large bandwidth requirement to the system. As a result, in the baseline implementation, only around 11% of the GPU computing capacity is utilized.

Therefore, in the new implementation, we adopt the strategy of computing 18 out of the 20 auxiliary vectors in run time rather than accessing them in global memory. Only the Coriolis force and the topographic data are stored as auxiliary vectors. Although the total computing amount is increased significantly, the actual processing time for a 1,024×1,024 sub-block is reduced from 117 ms to 48 ms,

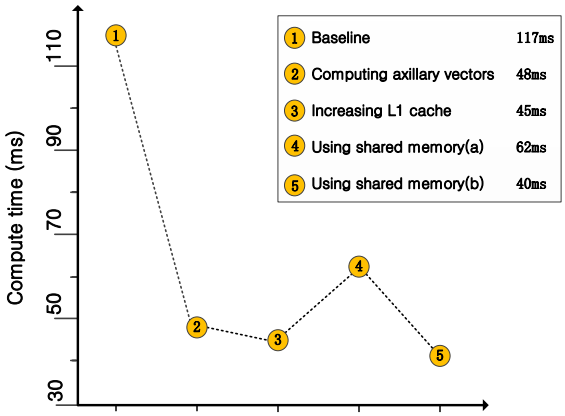

Figure 13. Comparison among different optimization techniques applied in the GPU stencil kernels.

shown as the second point in Fig. 13. Computing most auxiliary vectors in run time leads to great improvement of performance due to the substantial reduction of memory bandwidth requirements on the GPU side.

4.2.3 Increasing L1 cache

To further reduce the cost of the memory access, we can choose either to increase the size of the L1 cache or to use the shared memory. By increasing the size of the L1 cache from the default 16 KB to 48 KB, we further reduce the computation time from 48 ms to 45 ms, shown as the third point in Fig. 13.

4.2.4 Using shared memory

We can also move some of the frequently-used (especially the ones used by different threads in the same block) data into the shared memory to further improve data reuse. Due to the memory hierarchy of CUDA, shared memory can be accessible to all threads in the same block and temporarily store data during the lifetime of corresponding thread block. Compared with the global memory, the shared memory provides a much higher memory bandwidth and enables data reuse among all the threads in the same block. Applying a similar idea to the 3D stencil work by P. Micikevicius [15], we can load all the points in a thread block and the corresponding halos into the shared memory at the beginning of the computation, so that all the threads can reuse the neighboring points in the computation afterwards. For computing the 13-point stencil as shown in Fig. 3, each point in the middle of the block can be reused by 8 threads on average.

However, by adding the technique for using shared memory into our previous version, instead of achieving a better performance, the computation time for a 1,024×1,024 sub-block increases to 62 ms, shown as the forth point in Fig. 13. This is later found out to be a result of a low occupancy of the SM. In a 16 by 16 thread block, we need 12 KB shared memory to store all the points of the thread block and the halos. As the SM is under the configuration of 48-KB L1 cache and 16-KB shared memory, we can only support one thread block in one SM. By switching to the configuration of 16-KB L1 cache and 48-KB shared memory, we lose some cache hits but can then support more thread blocks in one SM. After switching the configuration, the shared memory version provides a computation time of 40 ms, shown as the fifth point in Fig. 13.

4.3 Optimization of the CPU stencil code

As shown in Fig. 7, a number of operations including halo updating, data copying, halo interpolating, stencil computing and data trans-

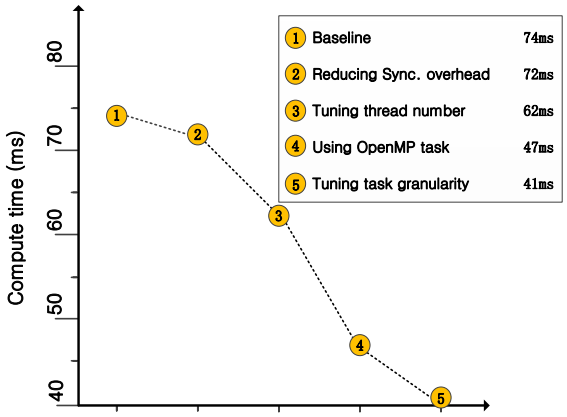

Figure 14. Comparison among different threading optimization techniques applied in the CPU stencil kernels.

ferring between CPUs and GPUs need to be performed on the CPU side. Among them, the most time-consuming part is to compute the stencils for the outer part within each sub-block. To improve the performance of stencil computation on the CPU side, we follow the idea from Datta et al. [5] by using array padding and in-loop vectorization techniques that help increase the in-core computing efficiency. In addition to that, we investigate several threading optimization techniques to maximize the computing throughput of the multi-core CPUs in each computing node.

As a starting point, we divide the interior area of the outer parts into four chunks along the four boundaries of the sub-block and parallelize them with separate parallel regions. To avoid false sharing and to reduce scheduling overhead, the OpenMP *for* worksharing construct is invoked at the outer loop of the stencil computation with static scheduling strategy over 12 threads. In addition to that, thread affinity is introduced to minimize the overhead of context switching. As mentioned in 4.2.2, instead of accessing the 20 auxiliary vectors, we compute 18 of them in run time so that the memory bandwidth requirement is substantially reduced. For our baseline CPU code, 74ms are needed to process the steps from ① to ⑧ in Fig. 7 for each stencil cycle, shown as the first point in Fig. 14. In the test, the sub-block size is 1,156×1,156 and the width of the outer part is adjusted to 66.

In the baseline implementation of the optimized hybrid algorithm, there are four barriers after processing each of the four interior chunks of the outer part, as seen in Fig. 7. The first three barriers are responsible for unexpected synchronization overhead, and can be eliminated by using the OpenMP *nowait* clause. Shown in Fig. 14 as the second point, by eliminating the three implicit synchronizations, the time cost is reduced from 74ms to 72ms.

Although most auxiliary vectors are calculated in run time in stead of being accessed from memory, there are still thirteen 3-component variables and two auxiliary coefficients needed during each 13-point stencil calculation, which may lower the computing throughput. To effectively hide the memory access latency, we increase the number of threads to twice as the number of CPU cores and the time cost is further reduced to 62ms, shown in Fig. 14 as the third point.

Since the four interior chunks of the outer part exhibit different data layouts (i.e., vertical v.s. horizontal) which might lead to load imbalance between threads, we try to schedule the threads dynamically and to reduce task granularity for each thread in order to maintain good load balance. However, no improvement has been observed. We then employ the OpenMP *task* construct [1] to achieve better scheduling efficiency. As a result, the time cost is reduced to 47ms, shown as the fourth point in Fig. 14. To further balance the trade-off between scheduling overhead and load imbalance, we search for the best task granularity for different chunks of the outer part and the time cost is eventually reduced to 41ms, shown as the fifth point in Fig. 14. Here we find the optimal number of threads is 12, which is equal to the number of CPU cores in each computing node of the Tianhe-1A.

5. Parallel Performance and Analysis

In this section, we first present numerical results on a model problem to validate the code and then conduct a more realistic simulation using real topographic data. Both strong and weak scaling results on the Tianhe-1A are presented and comparisons among the CPU-only and the hybrid approaches are provided.

To conduct an accurate performance measurement of our hybrid application that uses both CPUs and GPUs, we count the number of double-precision arithmetic operations in the code using three different methods:

- A manual count of the double-precision arithmetic operations in the code. We count "$+, -, \times, \div$" as one flop, while counting trigonometric computations as five.
- A direct measurement by running the CPU version of the code (the GPU version uses identical code for computing) with Performance API (PAPI) [4].
- An estimate of the double-precision arithmetic operations based on analysis of the GPU assembly code generated from the CUDA tool "cuobjdump".

The second and the third methods provide almost identical flop count of the application, while the result of the first method (manually counting) is around 10% higher. This is possibly due to compiler optimizations. In our study, we employ the second method (measurement by PAPI) to analyze the performance of our code.

5.1 Model validation and simulation results

We start the numerical tests from a model problem, zonal flow over an isolated mountain, which is taken from the benchmark test set of Williamson et al. [29]. In this test, a geostrophically steady-state flow impinges from west to east over a compactly supported mountain of conical shape. Fig. 15 shows the surface level distribution of

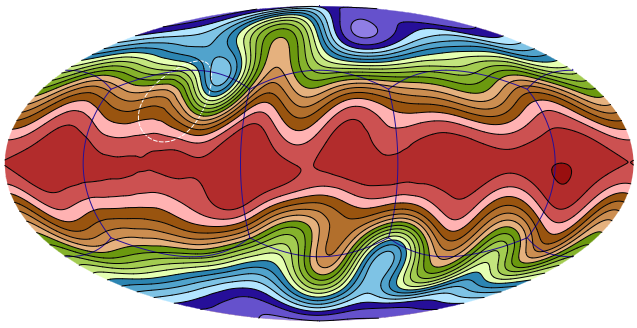

Figure 15. Surface level distribution of the atmosphere at day 15 in the isolated mountain test. Results are obtained on a 10,240×10,240×6 cubed-sphere mesh using 1,536 nodes of the Tianhe-1A. The conical mountain is outlined by the dotted circle in the figure.

the atmosphere at day 15 using a 10,240×10,240×6 cubed-sphere mesh (around 1-km resolution) on 1,536 nodes of the Tianhe-1A. In the figure, a Rossby-type gravity wave propagates all around the

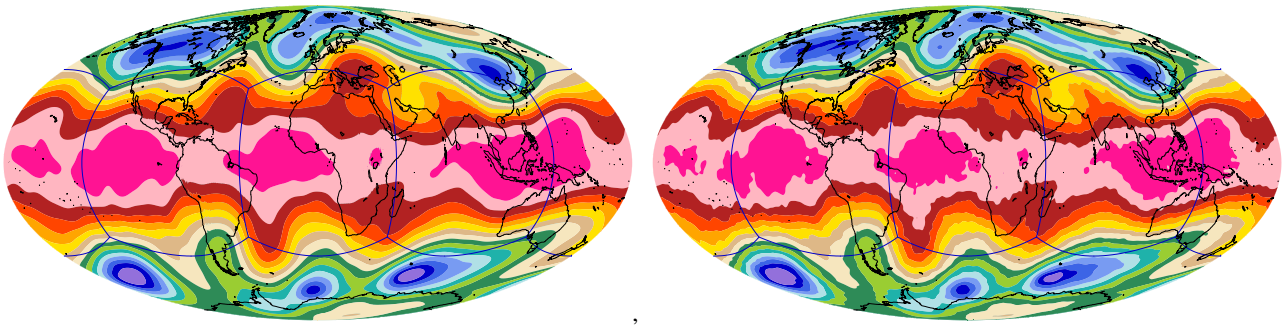

Figure 16. Surface level distribution of the atmosphere at day 15 in the real-topography test. We compare results at a 40-km resolution (left panel) and a 1-km resolution (right panel).

globe due to the presence of the mountain, which is in good agreement to published results (e.g., [12]).

We then conduct a more realistic simulation of the global atmosphere by inputing real topographic data of the Earth. The initial condition in this test is similar to that of the isolated mountain except that the surface level of the atmosphere is raised to avoid negative flow thickness due to the high elevations of some mountain ranges such as the Himalayas. Comparisons between results at a low-resolution of 40 km and a high-resolution of 1 km are given in Fig. 16, which clearly shows that as the resolution becomes finer, more details at small scales are discovered in the simulation.

5.2 Weak scaling results

In the weak scaling tests, we fix the mesh size on each sub-block to be 1,024×1,024 and then run the tests with different numbers of computing nodes (MPI processes). Configurations of the mesh size and the peak performance of available CPUs and GPUs in the Tianhe-1A are listed in Table 1. As the number of computing nodes increases from 6 to 3,750, the total number of unknowns is raised from 18.8 millions to nearly 12 billions.

Table 1. Configurations for the weak scaling tests.

Number of nodes	Mesh size	Peak of (CPU, GPU)
$6 = 6 \times 1 \times 1$	$6 \times 1024 \times 1024$	$(0.8, 3.1)$ Tflops
$24 = 6 \times 2 \times 2$	$6 \times 2048 \times 2048$	$(3.3, 12)$ Tflops
$96 = 6 \times 4 \times 4$	$6 \times 4096 \times 4096$	$(14, 49)$ Tflops
$384 = 6 \times 8 \times 8$	$6 \times 8192 \times 8192$	$(54, 198)$ Tflops
$864 = 6 \times 12 \times 12$	$6 \times 12288 \times 12288$	$(121, 445)$ Tflops
$1536 = 6 \times 16 \times 16$	$6 \times 16384 \times 16384$	$(216, 791)$ Tflops
$2400 = 6 \times 20 \times 20$	$6 \times 20480 \times 20480$	$(337, 1236)$ Tflops
$2904 = 6 \times 22 \times 22$	$6 \times 22528 \times 22528$	$(408, 1496)$ Tflops
$3750 = 6 \times 25 \times 25$	$6 \times 25600 \times 25600$	$(527, 1931)$ Tflops

In Fig. 17, we show the weak scaling performance of the proposed algorithms tested on the Tianhe-1A. In the tests, we compare four different algorithms, namely: the single-core CPU-only approach, the CPU-only approach with multi-threading, the hybrid CPU-GPU approach, and the optimized hybrid CPU-GPU approach with the adjustable partition between CPUs/GPUs and the "pipe-flow" scheme for communication-computation overlap. Except for the first approach in which there is no multi-threading, all the 12 CPU cores in each computing nodes are utilized in the tests.

As shown in the figure, not surprisingly, without using multi-threading, the CPU-only approach provides the lowest aggregate performance, which is improved by about 9.95 times when multi-threading is turned on. The hybrid CPU-GPU approach sustains an

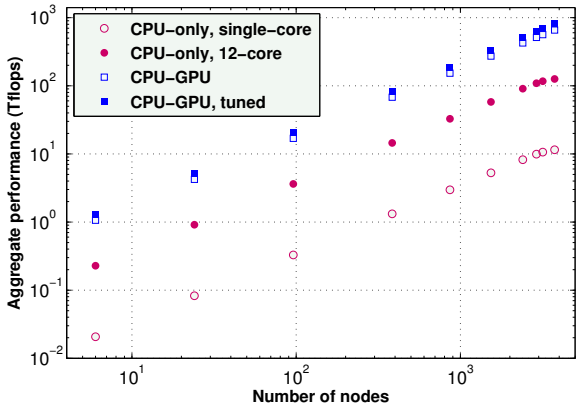

Figure 17. Weak scaling results on the Tianhe-1A. Shown in the figure are aggregate performances obtained by using: (○) the single-core CPU-only approach, (●) the multi-threaded CPU-only approach, (□) the hybrid CPU-GPU approach, and (■) the optimized hybrid CPU-GPU approach.

aggregate performance of about 658 Tflops, which is around 56 times better than that of the single-core CPU-only approach. When the optimized hybrid CPU-GPU approach is employed, thanks to the adjustable partition between CPUs and GPUs and the "pipe-flow" scheme for communication-computation overlap, the aggregate performance is further improved to 809.7 Tflops, which is around 32.8% of the peak performance.

The weak scalability of the CPU-only approach, no matter whether multi-threading is turned on or off, slightly suffers from higher communication overhead when larger number of nodes are utilized. On the other hand, the hybrid CPU-GPU approaches are able to achieve nearly ideal parallel efficiency even when the number of nodes is exceedingly large. This is because communications are totally hidden behind computations on the GPU side in the two hybrid approaches.

5.3 Strong scaling results

In the strong scaling tests, we fix the total problem size and increase the number of computing nodes. The total compute time should decrease as more computing nodes are utilized. However, because the computation-communication ratio becomes smaller at a higher node count, which will eventually affect the scalability, ideal strong scaling results are hard to obtain. Even a small fluctuation in the MPI communication will degrade the performance in the strong

scaling tests. Therefore, successful communication-computation overlap is a key to achieve expected scaling results.

Number of nodes	384	1536	2400	3750
Time (s)	56.9	14.4	9.4	5.9
Efficiency	1.00	0.99	0.97	0.98
Agg. Tflops	84.5	335.1	513.4	809.6

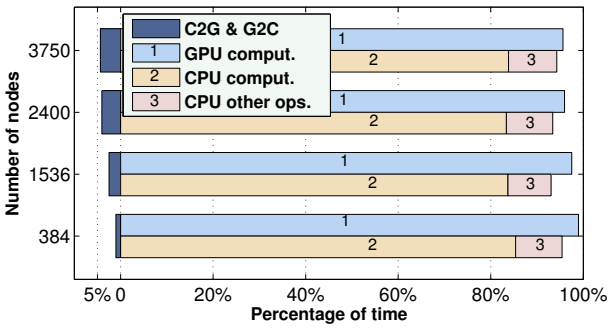

Figure 18. Strong scaling performance of the optimized hybrid CPU-GPU approach. The test results are obtained by running the code for 100 time steps using a 25,200×25,200×6 mesh on the Tianhe-1A. The upper table shows the parallel efficiency and the aggregate performance in the strong scaling tests. The lower figure provides a breakdown of elapsed time.

We run the strong scaling tests using a 25,200×25,200×6 mesh on the Tianhe-1A. Fig. 18 shows the strong scaling performance of the optimized hybrid CPU-GPU approach. For each node count, we calculate the averaged mesh size on each computing node and assign a proper portion of the outer layers to CPUs so that optimal performance can be achieved. As indicated in the top table of Fig. 18, nearly ideal strong scaling efficiency is sustained and an aggregate performance of 809.6 Tflops is delivered when 3,750 nodes are utilized.

There is only a very slight loss of scalability in the strong scaling tests, for which the reason is analyzed in the bottom figure of Fig. 18. In the figure, we can see that the computations as well as other operations (including interpolations and buffer preparations) on the CPU side (marked as "2" and "3" respectively) are totally hidden behind the computations on the GPU side (marked as "1"). The only part in the algorithm that could not be overlapped is the data transfer between CPUs and GPUs. Although this part only takes a small portion (less than 5%) of the total computing time, it still affects the parallel efficiency at larger node counts. We would like to point out here that both the parallel efficiency and the aggregate performance in the strong scaling results are superior to those in [26], in which communications were not totally overlapped with computations and the computing capacities from the CPUs were not fully exploited.

6. Concluding remarks

In this paper, we present a peta-scalable CPU-GPU algorithm for global atmospheric simulations. The major contributions of our work include: (1) an adjustable partitioning method that makes an equally-efficient utilization of both CPUs and GPUs in a heterogeneous system; (2) a "pipe-flow" communication scheme that conducts balanced and conflict-free message passing on the cubed-sphere geometry; and (3) systematic optimizations of the GPU and CPU codes that provide excellent performance for double-precision atmospheric simulation. With the above techniques combined, we manage to achieve nearly-ideal strong and weak scalabilities (both over 98%) on the Tianhe-1A. Our largest run sustains a performance of 0.8 Pflops in double precision to solve 12-billion unknowns using 3,750 nodes (45,000 cores and 3,750 GPUs) of the petascale system.

According to the strong and weak scaling results presented in the paper, the communication is totally hidden by the computation on CPUs and GPUs. Therefore, the proposed hybrid algorithm is expected to maintain a similar level of scalability when running with an even larger number of nodes on the Tianhe-1A. We also remark here that: (1) the basic idea of the adjustable CPU-GPU partition is not only applicable to other heterogeneous systems with different processor/accelerator ratios, but also extendable to a homogenous many-core system by redefining the partition; (2) although the new "pipe-flow" communication scheme is specifically designed for the cubed-sphere for conflict-free communication, similar ideas can be applied to other semi-structured geodesic meshes (e.g., [23, 28]) that are also becoming increasingly popular in global atmospheric modeling.

Acknowledgments

This work was partially supported by NSF China under grants 91130023 & 61170075, by 973 and 863 Programs of China under grants 2010CB951903, 2011CB309701 & 2010AA012301, and by Specialized Research Fund for State Key Laboratories in China. We would like to thank Xiao-Chuan Cai for insightful discussion on scalable algorithms in climate modeling, thank Paulius Micikevicius for giving suggestions on optimizing the CUDA kernel, and thank NSCC-TJ for providing access to the Tianhe-1A. This research received loads of supports from Guoxing Yuan, Canquan Yang, Guang Suo, Min Xie, Juan Chen, Xiangfei Meng, Jinghua Feng and Bin Xu. The authors are also grateful to the anonymous reviewers for valuable comments that have greatly improved the paper.

References

[1] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3): 404–418, march 2009.

[2] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. *PETSc Users Manual*. Argonne National Laboratory, 2010.

[3] M. Bernaschi, M. Bisson, T. Endo, S. Matsuoka, M. Fatica, and S. Melchionna. Petaflop biofluidics simulations on a two million-core system. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 4:1–4:12, New York, NY, USA, 2011. ACM.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[6] S. Gottlieb, C.-W. Shu, and E. Tadmore. Strong stability preserving high-order time integration methods. *SIAM Review*, 43:89–112, 2001.

[7] T. Hamada and K. Nitadori. 190 TFlops astrophysical N-body simulation on a cluster of GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[8] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 TFlops hierarchical N-body simulations on GPUs with

applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[9] K. Hamilton and W. Ohfuchi, editors. *High Resolution Numerical Modelling of the Atmosphere and Ocean*. Springer, 2008.

[10] T. Henderson, J. Middlecoff, J. Rosinski, M. Govett, and P. Madden. Experience applying Fortran GPU compilers to numerical weather prediction. In *Proceedings of 2011 Symposium on Application Accelerators in High Performance Computing (SAAHPC 2011)*, pages 34–41, 2011.

[11] Q. Hu, N. A. Gumerov, and R. Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 36:1–36:12, New York, NY, USA, 2011. ACM.

[12] R. Jakob-Chien, J. J. Hack, and D. L. Williamson. Spectral transform solutions to the shallow water test set. *J. Comput. Phys.*, 119:164–187, 1995.

[13] A. Kageyama and T. Sato. Yin-Yang grid: An overset grid in spherical geometry. *Geochem. Geophys. Geosyst.*, 5, 2004.

[14] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–7, 2008.

[15] P. Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. In *Proc. 2nd Workshop on General Purpose Processing on Graphic Processing Units*, pages 79–84, 2009.

[16] H. Miura, M. Satoh, T. Nasuno, A. T. Noda, and K. Oouchi. A Madden-Julian Oscillation event realistically simulated by a global cloud-resolving model. *Science*, 318:1763–1765, 2007.

[17] S. Osher and S. Chakravarthy. Upwind schemes and boundary conditions with applications to Euler equations in general geometries. *J. Comput. Phys.*, 50:447–481, 1983.

[18] W. M. Putman. *Development of the finite-volume dynamical core on the cubed-sphere*. PhD thesis, The Florida State University, 2007.

[19] W. M. Putman and M. Suarez. Cloud-system resolving simulations with the NASA Goddard Earth Observing System global atmospheric model (GEOS-5). *Geophys. Res. Lett.*, 38, 2011.

[20] C. Ronchi, R. Iacono, and P. Paolucci. The cubed sphere: A new method for the solution of partial differential equations in spherical geometry. *J. Comput. Phys.*, 124:93–114, 1996.

[21] J. A. Rossmanith. A wave propagation method for hyperbolic systems on the sphere. *J. Comput. Phys.*, 213:629–658, 2006.

[22] R. Sadourny. Conservative finite-difference approximations of the primitive equations on quasi-uniform spherical grids. *Mon. Wea. Rev.*, 100:211–224, 1972.

[23] R. Sadourny, A. Arakawa, and Y. Mintz. Integration of the nondivergent barotropic vorticity equation with an icosahedral-hexagonal grid for the sphere. *Mon. Wea. Rev.*, 96:351–356, 1968.

[24] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[25] T. Shimokawabe, T. Aoki, J. Ishida, K. Kawano, and C. Muroi. 145 TFlops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction. *Procedia Computer Science*, 4: 1535 – 1544, 2011. Proceedings of the International Conference on Computational Science (ICCS 2011).

[26] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 3:1–3:11, New York, NY, USA, 2011. ACM.

[27] S. Shingu, H. Takahara, H. Fuchigami, M. Yamada, Y. Tsuda, W. Ohfuchi, Y. Sasaki, K. Kobayashi, T. Hagiwara, S.-i. Habata, M. Yokokawa, H. Itoh, and K. Otsuka. A 26.58 Tflops global atmospheric simulation with the spectral transform method on the Earth Simulator. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC '02)*, pages 1–19, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[28] D. L. Williamson. Integration of the barotropic vorticity equation on a spherical geodesic grid. *Tellus*, 20:642–653, 1968.

[29] D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, and P. N. Swarztrauber. A standard test set for numerical approximations to the shallow water equations in spherical geometry. *J. Comput. Phys.*, 102:211–224, 1992.

[30] M. Xie, Y. Lu, K. Wang, L. Liu, H. Cao, and X. Yang. The Tianhe-1A interconnect and message passing services. *IEEE Micro*, 1, 2012.

[31] C. Yang and X.-C. Cai. Parallel multilevel methods for implicit solution of shallow water equations with nonsmooth topography on the cubed-sphere. *J. Comput. Phys.*, 230:2523–2539, 2011.

[32] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su. The Tianhe-1A supercomputer: Its hardware and software. *J. Comput. Sci. Tech.*, 26, 2011.