Developing an ELM Ecosystem Dynamics Model on GPU with OpenACC *

Peter Schwartz[1][0000−0002−0852−5528], Dali Wang[1][0000−0001−6806−5108], Fengming Yuan[1][0000−0003−0910−5231], and Peter Thornton[1][0000−0002−4759−5158]

Environmental Sciences Division, Oak Ridge National Laboratory, Oak Ridge TN 37830, USA
{schwartzpd,wangd,yuanf,thorntonpe}@ornl.gov

Abstract. Porting a complex scientific code, such as the E3SM land model (ELM), into a new computing architecture is challenging. The paper presents design strategies and technical approaches to develop an ELM ecosystem dynamics model with compiler directives (OpenACC) on Nvidia GPUs. Code has been refactored with advanced OpenACC features (such as deepcopy and routine directives) to reduce memory consumption and to increase the levels of parallelism through parallel loop reconstruction and new data structures. As a result, the optimized parallel implementation achieved more than a 140-time speedup (50 ms vs 7600 ms), compared to a naive implementation on a single Nvidia V100. On a fully loaded computing node with 44 CPUs and 6 GPUs, the code achieved over 3.5-time speedup, compared to the original code on CPU. Furthermore, the memory footprint of the optimized parallel implementation is 300 MB, which is around 15% of the 2.15 GB memory consumed by a naive implementation. This study is the first effort to develop ELM component on GPUs efficiently.

Keywords: Earth System Models · Exascale Energy Earth System Model · E3SM Land Model · OpenACC · Functional Unit Testing · Ecosystem Dynamics

1 Introduction

The Exascale Energy Earth System Model (E3SM) is a fully coupled Earth system model that uses code optimized for the DOEś advanced computers to address the most critical scientific questions facing our nation and society [4]. The E3SM contains several community models to simulate major Earth system components: atmosphere, ocean, land, sea ice, and glaciers. Within the E3SM framework, the

* This research was supported as part of the Energy Exascale Earth System Model (E3SM) project, funded by the U.S. Department of Energy, Ofce of Science, Ofce of Biological and Environmental Research. This research used resources of the Oak Ridge Leadership Computing Facility and Experimental Computing Laboratory at the Oak Ridge National Laboratory, which are supported by the Ofce of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

E3SM Land Model (ELM) is designed to simulate how the changes in terrestrial land surfaces interact with other Earth system components and has been used to understand hydrologic cycles, biogeophysics, and ecosystem dynamics[3]. ELM software has several distinguishing computational features: 1) all the biophysical and biochemical processes are simulated on individual land surface units (i.e., gridcell) independently; 2) highly customized globally accessible data structures are used to represent the heterogeneity of Earthś landscape; 3) all subroutines are computationally trivial[10].

Most current high-end supercomputers use heterogeneous hardware with accelerators[2]. The E3SM, consisting of millions of lines of legacy code developed for traditional homogeneous multicore processors, cannot automatically benefit from the advancement of these supercomputers. Refactoring and optimizing the E3SM models for new architectures with accelerators is challenging but inevitable.

Since rewriting a large-scale legacy code in a new programming language (such as CUDA) is not practical, two general approaches (compiler directives and use of GPU-ready libraries) have been adapted to develop E3SM code for computing systems with accelerators. For example, Kokkos libraries has been used to increase the performance on the E3SM atmosphere model on Nvidia GPUs on Summit[1], OpenACC and Athread have been used to develop the community atmosphere model on manycore 64-bit RISC processors in Taihu Light[9], and OpenACC has been used to accelerate MPAS microphysics WSM6 model on one Nvidia GPU[5].

With the availability of high-resolution atmospheric forcing (such as temperature, precipitation, shortwave radiation, and vapor pressure) [6] and land surface proprieties (such as vegetation and soil properties maps), it is desirable to conduct high-fidelity land simulations with ELM at 1km-resolution to deliver a "gold standard" set of results describing the surface weather and climate, as well as the energy, water, carbon, and biogeochemistry processes at a continental scale. The ultra high resolution ELM simulations over North America covers a landscape of 24 million gridcells, 350 million columns, and 700 million vegetation types, is only feasible with highly efficient use of the accelerators within high-end supercomputers. Furthermore, for the reason that there are over one thousand of subroutines in ELM, the majority of which are computationally simplistic, using compiler directives is the appropriate approach to accelerate ELM onto GPU systems. The paper reports a development of an ecosystem dynamics model within ELM on Nvidia GPU using OpenACC.

1.1 Computational platform and software environment

The computational platform used in the study is the Summit leadership computing system at the Oak Ridge National Laboratory. Summit has 4,608 computing nodes, most of them contain two 22-core IBM POWER9 CPUs, six 16-GB NVIDIA Volta GPUs, and 512 GB of shared memory. Technically, this study uses

42 CPU cores (2 CPU cores are reserved for system functions) and all the non-tensor cores in GPUs. Software environment used in our study include NVIDIA HPC 20.11 and several libraries: openMPI (spectrum-mpi/10.4.0.3-20210112), netcdf (netcdf-c/4.8.0, netcdf-fortran/4.4.5), pnetcdf(1.12.2), HDF (1.10.7), and CUDA (11.1).

2 Method

The ecosystem dynamics model simulates the biogeochemical cycles of the ecosystem, including carbon, nitrogen, and phosphorus. It contains many function groups, such as nitrogen deposition/fixation, maintenance and growth respiration, phosphorus deposition, and soil litter decomposition. The ecosystem dynamics model is the most sophisticated model within ELM that contains over 90 subroutines and accesses over 2000 globally accessible variables, many of them 3D arrays.

We have developed a Functional Unit Testing (FUT) framework to generate standalone ELM models to accelerate code porting and performance tuning. The FUT is a python toolkit built upon the previous software system designed to facilitate scientific software testing[8, 7]. In this study, a standalone ecosystem dynamics model is first generated for code porting and performance evaluation. To quickly assess the performance of ELM, we have generated synthesized data using the observational data from AmeriFlux (ameriflux.lbl.gov) as the forcing data set for the code development. The forcing data and the global variables for this model take 11GB GPU memory using 6000 gridcells (approximate 1.8MB data per gridcell). Since each Nvidia V100 contains 16 GB of memory, 5GB shared GPU memory is available for other globally accessible variables, history data, external forcing data, and ELM kernels. The ecosystem dynamics model takes an hourly timestep. An ELM spin-up simulation generally covers a period of 800 to 1000 years, and an ELM transit simulation runs over a period of 100-200 years, therefore, the ecosystem dynamics model usually are executed around 8-10 million times on each gridcell.

2.1 Function groups inside the ecosystem dynamics model

The ecosystem dynamic model is the most complex submodel within ELM containing four complex subroutines (initialization, leaching, noleaching1, and noleaching2). These four complex subroutines contain over 90 subroutines and access around 2000 global variables. From the function perspective, the ecosystem dynamics model can be grouped into several modules, such as carbon, nitrogen, and phosphorus dynamics, phenology, soil litter, gap mortality, and fire (Fig. 1). For a better illustration, many secondary and supporting functions, such as IO, timing, and other utility functions are not shown in the picture.

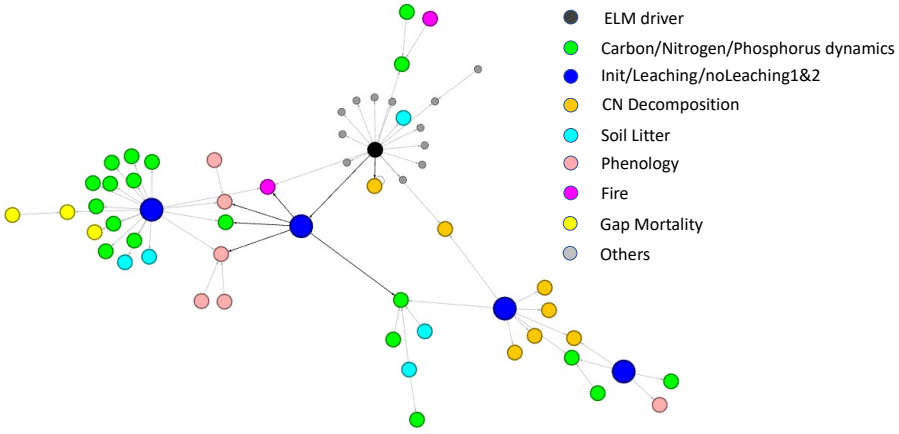

Fig. 1: The ecosystem dynamics model contains several function modules. There are 129 nodes, each represents a subroutine. The four blue node represent the complex subroutines: initialization, leaching, noleaching phase 1, noleaching phase 2.

2.2 Performance comparison and a naive OpenACC implementation

This study adopts a node-level performance comparison, in which a similar full workload is placed onto all the CPU cores or all GPUs within a single node, and the execution time of the original CPU code and GPU implementation at a single hourly simulation timestep are collected for performance evaluation. Specifically, on a single Summit node, we assign 36000 gridcells on GPUs (6000 gridcells on each GPU) and 36036 gridcells on CPUs (858 gridcells on each CPU core). With the workload of 858 gridcell, the original CPU-based ecosystem dynamics model takes 150 millisecond (ms) to finish a single timestep (hourly) of simulation.

The original code organizes gridcells into clumps on each CPU core and the ecosystem dynamic model runs over these clumps of gridcells. Therefore one of the simplest implementations is to instrument OpenACC directives into the original code and onto these existing loops. Specifically, this implementation contains four major steps: 1) use data directive to create a data region and copy all the input data into the data region once at the beginning of simulation; 2) use routine directives to generate GPU kernels of the majority of ecosystem dynamics subroutines, including three complex subroutines, Leaching, NoLeaching Phase 1, and NoLeaching Phase 2, as well as many subroutines inside them; 3) place parallel loop construct to loop over the gridcells, then 4) launch these GPU kernels in parallel (Fig. 2). This naive implementation does work but comes with very poor performance (over 7 seconds) and consumes a large amount of memory (2.15 GB) for just the kernel.

Algorithm 1 A Naive Implementation

```
if first step then
    acc enter data copyin                    ▷ Start Unstructured Data Region
acc parallel loop default(present)
for each gridcell do
    GET BOUNDS OF GRIDCELL                   ▷ Bounds holds subgrid info
    LEACHING
    NOLEACHING PHASE 1
    NOLEACHING PHASE 2
```

Fig. 2: All subroutines were ported using acc routine directive and parallelized across the existing gridcell loop.

2.3 Code Optimization

Several optimizations were developed to improve the code performance, such as reducing the memory consumption, restructuring parallel loops, deploying reduction clause, and increasing parallelism over independent elements.

Reducing the memory allocation of local variables Each Nvidia V100 has 16 GB of memory to contain all the data and ELM kernels. Memory allocation operations on GPU are more expensive than those on the host, so we need to reduce the memory consumption of each kernel. In this study, we deployed many methods to reduce the size of these local variables, such as converting arrays into scalars and compressing the sparse arrays into dense arrays. For example: In SoilLittVertTransp, there were 9 local arrays that would be allocated with a size of 238 doubles for each gridcell (that is 17136 (9*238*8) bytes in total). After refactorization, these arrays were replaced with a couple of 64-bit scalar and 4 dense arrays (less than 550 bytes in total). This memory saving is tremendous since we want to put up to 6000 gridcells on a single GPU and decreases the total time of ecosystem dynamics model to 150ms from 7600ms.

Restructure parallel loops The routine directive provides the ability to quickly test functions on the GPU, but for subroutines with internal loop structures and nested function calls, performance degradation is expected. In the ecosystem dynamics model, routines that loop over the same subgrid element (column or patch) tend to be clustered together. For example, Fig. 3 shows a group of functions labelled as SetValues, where the vegetation and column subroutines compute over patch and column subgrid arrays, respectively. A reliable optimization strategy is to refactor these routines to remove their internal loops and change the external gridcell based loops to the relevant subgrid element, which allows them to be grouped under different parallel loop constructs. In this case, each subroutine is actually completely independent with the others so

Algorithm 2 Parallelize Gridcell	**Algorithm 3** Parallelize Subgrid
acc parallel loop **for** each gridcell **do** VEGCFSETVALUES(array) VEGNFSETVALUES(array) VEGPFSETVALUES(array) COLCFSETVALUES(array) COLNFSETVALUES(array) COLPFSETVALUES(array)	**acc parallel loop async** **for** each subgrid element **do** VEGCFSETVALUES(index) ⋮ **acc parallel loop async** **for** each subgrid element **do** COLPFSETVALUES(index)

Fig. 3: The SetValues routines were refactored to remove the internal loops. In the original code, there are a series of similar operations to set values for a group of global variables with different filters at each gridcell. After the refactorization, these SetValues operations are placed into a series of inner loops that can be launched asynchronously

that the SetValues group of functions can utilize asynchronous kernel launch. The loop re-factorization decreased the execution time from 30 ms to 16 ms and proves that the routine directive can provide a significant speedup over the CPU implementation.

Accelerate Internal Loops For subroutines that must have their internal loops, we forego the routine directive and deploy different parallel techniques specific to each loop. One good example is the fire module that includes two large subroutines (FireFluxes and FireArea), and each also contains many subroutines. When deploying the OpenACC routine directive within the fire module, a single step execution takes around 50 ms.

Algorithm 4 CPU Code	**Algorithm 5** GPU Code

for each Soil Patch **do**
for each Soil Level **do**
Get: Column from patch
Sum: patch vars to column
	acc parallel loop gang worker
	for each Soil Level **do**
	for each Soil COL **do**
	Init: sums
	acc vector reduction(+:sums)
	for each patch in COL **do**
	Reduce: patch vars to column
	output ← *sums*

Fig. 4: CPU Code(left) only uses two loops, the GPU code(left) required an additional loop to prevent race conditions during reduction. The achieved speedup is over 2x.

The ecosystem dynamics model uses hierarchical data structures and the lower level variables (i.e. patch level variables) have to be aggregated into corresponding higher level variables (i.e. column level variables). Fig. 4 shows an example of aggregating variables from patch level to column level in the FireFluxes subroutine. The code is optimized by using gangs and workers to parallelize the two outer (collapsed) loops with an inner vector loop performing the reduction. The reduction operation is very efficient in our case because there are a maximum of 33 patches in each column and each warp of the Nvidia V100 has 32 threads. With the reduction, we can finish the operations in 0.3 ms. After the optimizations, the execution time of the entire GPU-based fire module was reduced to 4 ms, which includes 20 kernels.

Parallelize Over Nutrients or Output Variables To further improve the performance of the ecosystem dynamics model, we investigate the similar operations among nutrients, such as carbon, nitrogen, and phosphorus (CNP), and the output variables (such as temperature, water, sources and sinks of carbon/nitrogen/phosphorous). A new array of derived types is created with each nutrient element pointing to a set of three dimensional arrays corresponding to a CNP computation. This data structure is initialized and moved to the device only at the start of a run via deepcopy. We can then either collapse a CNP loop into the parallel loop construct or use the nutrient loop to asynchronously launch kernels to improve the code performance. A good example is the transport operation inside the SoilLittVertTransp subroutine (Fig. 5), where we take advantage of asynchronous compute to pipeline the creation of local arrays for tridiagonal coefficients.

Algorithm 6 Init Pointer List

```
Loop over C,N,P types
for each ntype do
    list[ntype]%conc → ntype%conc
    list[ntype]%src → ntype%src
    list[ntype]%trcr → ntype%trcr
acc copyin(list)
```

Algorithm 7 Parallelize Over CNP

```
for each nutrient type do
    acc parallel loop async(ntype)
    for each decomp source do
        for each soil levels do
            for each soil column do
                Set outputs using list
                Solve TRANSPORT
```

Fig. 5: (Left) An array of a derived types are created to global output variables and copied to device at start of run. (Right) The nutrient type loop is used to overlap the transport kernels.

The left panel of Fig. 5 illustrates the creation of the new derived-type, and the right panel shows the new asynchronous loop structure of the transport algorithm inside the SoilLittVertTransp module. The vertical transport among

soil levels in each soil column is calculated using a tridiagonal solver. The solving algorithm does not have enough dimensions to fully utilize the GPU resources, so we take advantage of the independence between CNP inputs and outputs to occupy the device. The asynchronous launching of these kernels allows the whole SoilLittVertTransp module to finish in 5 ms, compared to over 40 ms using the routine directive after removing excess memory allocations. This method of using arrays of pointers is also used to dramatically improve the parallelism of output modules, such as history buffer that average over 500 variables but is not a part of the ecosystem dynamics model.

3 Results

3.1 Overall Performance Improvement

We gather the subroutines in the ecosystem dynamics model into many groups and collect the execution time separately (Table1). The GPU-based ecosystem dynamics model achieves a 3.0 times speedup over the original code on CPU when the Summit node is fully loaded. The timing data of the majority (10 out of 15) function groups show good speedups (ranging from 2.3 to 8.3 times). Especially, gap mortality contains many global-variable operations (similar to SetValues) that have been refactored into parallel do loop and are launched asynchronously on GPU. Maintenance respiration contains two simple loops that have been refactored into parallel do loops with reduction clause. The execution times of the other 5 groups are relatively small (less than 2 ms). The slow down of these groups is mainly because the overhead associated with GPU kernels overshadowed the small computational part of these subroutines. After optimization, the entire ecosystem dynamic model (GPU kernels with all the nested subroutines) require around 300MB memory.

3.2 Profiling Details

To determine the limitations of OpenACC's routine directive and better understand performance improvements, we used Nvidia's Nsight Compute and Nsight Systems to collect GPU metrics and traces for SoilLittVertTransp, FireMod, and SetValues subroutines at various stages of optimization. Table 2 shows the metrics of the kernels that illustrate different shortcomings of prior versions the GPU kernels. While no one set of metrics will best describe the performance of different algorithms, the wallclock time, kernel launch overhead, and total instructions issued are used to frame performance discussion in this work. The overheads listed in Table 2 are the percentage of wallclock time spent not computing but on launching the kernel, allocating memory, and creating device streams (if applicable). Nsight Systems drastically increases initial kernel launch times but agrees with Nsight Compute in reporting the kernel compute time, so to consistently calculate overhead, we subtract the kernel compute time from the normal total wallclock time and take the percentage.

Table 1: Comparison of average execution time between GPU and CPU versions of the ecosystem dynamics model (single timestep) on a fully loaded single Summit node. Each node has the the same size grid divided between 6 GPUs or 42 CPUs

Function Group	GPU(ms)	CPU(ms)	Speedup
SetValues	15.49	46.16	2.98
NDeposition/Fixation	0.09	0.02	0.24
Respiration/PDeposition	0.33	2.77	8.30
Decomp. Rate	1.31	3.08	2.35
Vertical Decomp.	2.74	7.77	2.84
Alloc Phase 1	1.51	0.93	0.62
SoilLittDecomp 1	6.12	23.30	3.81
SoilLittDecomp 2	1.18	1.89	1.60
Phenology	2.26	0.58	0.26
Growth and Root Respiration	1.03	0.43	0.42
StateUpdate 1	2.79	12.21	4.39
SoilLittVertTransp	5.94	17.35	2.92
GapMortality	2.56	17.54	6.85
StateUpdate 2	1.88	4.95	2.64
FireMod	4.84	11.19	2.31
Total	50.05	150.2	3.00

SoilLittVertTransp The initial OpenACC implementation of SoilLittVertTransp had the worst performance of the ecosystem dynamics model and even the full E3SM Land Model runs. The reliance of the CPU version on dynamically allocated arrays can be distilled to the over-inflated number of instructions reported on the GPU, which is nearly three orders of magnitude larger than that with the arrays refactored out. The memory workload and warp state metrics further support that this memory is the bottleneck by showing very high L1 and L2 hit rates and related stalls (not listed in the paper).

After the initial refactoring (Mem Opt.), the performance is still 2.6 times slower than a single CPU core given a similar workload. While the kernel only uses half the threads in a warp (on average), the profilers actually report the kernel compute time as around 10ms. The overheads associated with launching the kernel contribute the most to the total time and the subroutine is simply too big and complicated to be ported using only sequential routine directives. The metrics for the newest OpenACC implementation illustrates much more efficient utilization of the GPU's warps and pipelines.

FireFluxes The initial implementation of OpenACC worked better for the FireFluxes subroutine, which consists exclusively of computations on global data types. However, the overheads are again a severe penalty due to the many nested loop structures and the hundreds of global variables that need to be passed as arguments by the kernel launcher. The final optimizations resulted in a great

increase in the required instructions, but being able to hide them with asynchronous kernel launches and saturating the GPUs resources yields very high efficiency and a little more than 2.3 speedup over the CPU.

Table 2: High-level metrics for three kernels showing differences between optimization methods. The routine directive can result in very high overheads, and poor memory allocations result in large excess in instructions issued. For FireFluxes, the excess instructions are due to extra loops but is alleviated by better saturation of GPU.

Kernel(ver)	**Time(ms)**	**Overhead(%)**	**Inst.(millions)**
SoilLittVertTransp			
Orig	10,000	49.8	34,500
Mem Opt	44.0	70.0	48.7
Final Opt	6.95	11.8	40.9
FireFluxes			
Orig.	45.0	92	9.83
Opt	3.67	27.9	197
SetValues			
Orig.	29.4	70.2	15.3
Opt.	15.4	11.6	14.5

SetValues The default OpenACC implementation of the SetValues functional group had a speedup around 1.5 times relative to the CPU, but the total time taken was significant relative to the other modules. The SetValues subroutines involved initializing hundreds of arrays of three derived-types at the subgrid patch and column level, which can all be done independently. While the OpenACC routine directive allows for multiple levels of parallelism, compiling as a sequential routine is the most reliable and free of compilation errors for our use case. The profiling data identified that kernel launch overhead was the bottleneck, and so the routines were simplified to as described in Section 2.3. The overhead is greatly reduced and mostly due to CUDA API calls to allocate memory on the host and device and create streams.

4 Conclusion and future work

The study reports several design and optimization strategies of developing an ecosystem dynamics model within ELM using compiler directives (OpenACC) on Nvidia GPUs. The routine directive and deepcopy capabilities of OpenACC provided a robust method for accelerating very complex module within ELM, with certain functions receiving immediate speedup. After a series of further code analyses and refactorization, the parallel GPU implementation (with a small memory footprint of 300 MB) achieved over 3.5 times speedup, compared to the original CPU code on a fully loaded Summit computing node. Computationally, ELM doesn't have a single submodel that dominates run time but around a dozen complex components that contribute more or less equally. Accelerating

the full code base requires a flexible approach that can handle different algorithms, and this work demonstrates methods for reworking algorithms and data structures in tandem with compiler directives to tackle exascale climate simulations. Future work will focus on parallelization of other models within ELM and further integrated performance tuning. For the ultra-high resolution ELM simulation over North America, if we assign 36000 gridcells to each node; the 24 million gridcells of North America require 680 nodes, which is around 15% of the total capability of the 4608-node Summit. With an estimated 3.0 times overall speedup, a 100-year simulation over North America takes around 2 days.

References

1. Bertagna, L., Guba, O., Taylor, M.A., Foucar, J.G., Larkin, J., Bradley, A.M., Rajamanickam, S., Salinger, A.G.: A performance-portable nonhydrostatic atmospheric dycore for the energy exascale earth system model running at cloud-resolving resolutions. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14. IEEE (2020)
2. Bourzac, K.: Supercomputing poised for a massive speed boost. Nature **551**(7680), 554–557 (2017)
3. Burrows, S., Maltrud, M., Yang, X., Zhu, Q., Jeffery, N., Shi, X., Ricciuto, D., Wang, S., Bisht, G., Tang, J., et al.: The doe e3sm v1. 1 biogeochemistry configuration: Description and simulated ecosystem-climate responses to historical changes in forcing. Journal of Advances in Modeling Earth Systems **12**(9), e2019MS001766 (2020)
4. Golaz, J.C., Caldwell, P.M., Van Roekel, L.P., Petersen, M.R., Tang, Q., Wolfe, J.D., Abeshu, G., Anantharaj, V., Asay-Davis, X.S., Bader, D.C., et al.: The doe e3sm coupled model version 1: Overview and evaluation at standard resolution. Journal of Advances in Modeling Earth Systems **11**(7), 2089–2129 (2019)
5. Kim, J.Y., Kang, J.S., Joh, M.: Gpu acceleration of mpas microphysics wsm6 using openacc directives: Performance and verification. Computers & Geosciences **146**, 104627 (2021)
6. Thornton, P.E., Shrestha, R., Thornton, M., Kao, S.C., Wei, Y., Wilson, B.E.: Gridded daily weather data for north america with comprehensive uncertainty quantification. Scientific Data **8**(1), 1–17 (2021)
7. Wang, D., Wu, W., Janjusic, T., Xu, Y., Iversen, C., Thornton, P., Krassovisk, M.: Scientific functional testing platform for environmental models: An application to community land model. In: International Workshop on Software Engineering for High Performance Computing in Science, 37th International Conference on Software Engineering (2015)
8. Wang, D., Xu, Y., Thornton, P., King, A., Steed, C., Gu, L., Schuchart, J.: A functional test platform for the community land model. Environmental Modelling & Software **55**, 25–31 (2014)
9. Zhang, S., Fu, H., Wu, L., Li, Y., Wang, H., Zeng, Y., Duan, X., Wan, W., Wang, L., Zhuang, Y., et al.: Optimizing high-resolution community earth system model on a heterogeneous many-core supercomputing platform. Geoscientific Model Development **13**(10), 4809–4829 (2020)
10. Zheng, W., Wang, D., Song, F.: Xscan: an integrated tool for understanding open source community-based scientific code. In: International Conference on Computational Science. pp. 226–237. Springer (2019)