Parallel Computing for Module-Based Computational Experiment [⋆]

Zhuo Yao[1], Dali Wang[2][0000−0001−6806−5108] [⋆⋆], Danial Riccuito[2][0000−0002−3668−3021], Fengming Yuan[2][0000−0003−0910−5231], and Chunsheng Fang[3]

[1] University of Tennessee, Knoxville, TN 37996, USA
`yaoz5@vols.utk.edu`
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37831, United States
`(wangd, ricciutodm, yuanfm)@ornl.gov`
[3] Jilin University, Changchun, Jilin, 130012, P.R. China
`fangcs@jlu.edu.cn`

Abstract. Large-scale scientific code plays an important role in scientific researches. In order to facilitate module and element evaluation in scientific applications, we introduce a unit testing framework and describe the demand for module-based experiment customization. We then develop a parallel version of the unit testing framework to handle long-term simulations with a large amount of data. Specifically, we apply message passing based parallelization and I/O behavior optimization to improve the performance of the unit testing framework and use profiling result to guide the parallel process implementation. Finally, we present a case study on litter decomposition experiment using a standalone module from a large-scale Earth System Model. This case study is also a good demonstration on the scalability, portability, and high-efficiency of the framework.

Keywords:
Parallel computing · scientific software · message passing based parallelization · profiling

1 Introduction

Scientific code that incorporates important domain knowledge plays an important role in answering essential questions. In order to help researchers understand and modify the scientific code using good approaches from best software development practices[4, 7], we prefer a framework to visualize code architecture and

⋆ This research was funded by the U.S. Department of Energy, Office of Science, Biological and Environmental Research (Terrestrial Ecosystem Sciences),Advanced Scientific Computing Research (Interoperable Design of Extreme-scale Application Software).

⋆⋆ Corresponding author, POBox 2008, MS 6301, Oak Ridge National Lab, Oak Ridge, TN 37831, USA. Tel: 8652418679 individual modules, so that researchers can conveniently use modules to design specific experiments and to optimize the code base. We hope the framework to be highly portable and multi-platform compatible, therefore scientists can use it on different platforms. At the same time, we would like the framework to offer concurrent data analysis interface, which decouples the analysis from the file-based I/O in order to facilitate the data analysis.

In the previous work [13], we introduced and designed a unit testing framework that isolates specific functions from complex software base and offers an in-situ data communication service [11]. This service runs by an analysis code locating remotely as the original scientific code is running. The testing driver can build a necessary environment which suits the needs of a typical experiment. With the framework, the scientists can track and manipulate variables between modules or inside modules to better meet their needs. However, the above-mentioned framework is unable to meet the requirement of many scientific applications that simulate large-scale phenomena with complex mathematical models on supercomputers. The examples of these scientific applications include large-scale models to predict climate change, air traffic control, power grids, and nuclear power plants. Therefore, improving the overall performance of the functional-unit testing platform for scientific code is significant.

2 Related Works

Scientific software can be bulky and complicated, it is important to analyze the crucial performance factor in order to optimize the code base. A diverse set of tools and methodology were used to identify the performance and scaling problems, including shell timers, library subroutines, profilers, and consisting of tracing analysis tools and sophisticated full-featured toolsets. For example, the shell timex reports system related information in a common format across a variety of shells. Profiling measures the frequency and duration of functions or memory and time complexity of a program through instrumenting program's source code or executable file. Unlike profiling, the tracing approach records all events of an application run with precise time stamps and many event type specific properties [2]. Performance analysis toolkits include three steps: instrumentation, measurement, and analysis. Among all popular toolkits, this paper chooses Vampir to visualize the Fortran program behavior, recorded by Score-P in open trace format.

There are some unit test frameworks available for Fortran: Fortran unit test framework (FRUIT), pFUnit, ObjecxxFTK, and FLIBS. This paper takes FRUIT as an example to describe the team's work and weakness. The FRUIT is written in FORTRAN 95, and it can test all FORTRAN features. FRUIT includes five features: assertions for different types of unit tests, function files used to write tests, summary reports of the success or failure of tests, a basket module to invoke set up and tear down functions, and driver generation. Set up and tear down functions are used to perform initialization and finalization operations to all tests within a module. However, all these tools never consider testing modules with groups of global variables. It is known that defining variables on the global scope is a bad but common practice in scientific software development. Extensive usage of global variables makes dependencies analysis difficult and complicates module loading, which in turn cause complicated module interactions.

Paper [3] introduced a Python-based tool called KGEN that extracts a part of Fortran code from an application. KGEN can extract specific statements into stand-alone kernels from large applications including MPI application, it can also provide a way to automatically generate a data stream to drive and verify the execution of the extracted kernel. The tool can deal with global variables and have parallel computation configuration, but it includes excessive time statistics and built-in libraries for kernel generation, which decrease the overall performance.

Paper [13] developed a platform which first split data and library dependencies over software modules and then drove the unit functions with the extracted data stream from original scientific code. With the verified platform, the scientific model builders can track interesting variables either in one single subroutine or among different subroutines. However, the research did not deal well with the performance issues related to long time scientific simulations.

3 System Design

In previous research, we focus on how to design a framework to generate unit testing and how to drive the unit testing and validate the correctness of the infrastructure. The framework adapts a serial computational model and does not consider the performance issues associated with a long period of time simulation, such as a 10-year period simulation at a half-hour timestep. Therefore, in this study, in order to make our kernel generation infrastructure more reliable and practical, we improve our design to embrace parallel computing methods.

First, we create an experiment with user-required subroutines. We extract specific unit modules with built-in logic based on user requirements. Then, we apply our sequential unit testing framework to isolate the user required modules and then validate the correctness of the framework.

Second, we design an efficient model to support large data transfer between different modules and continuous step-wise simulations. In our previous work, we divided data based on data access method into three groups: *write_only*, *read_only* and the *modify* [13]. In the parallel version, since disk I/O operation was time-costly, we switched the file-based I/O operations to memory read/write to improve the overall performance. In this paper, we used the code analyzer to analyze data flow based on module relevance. Then, we divided the variables into three groups (*In-Module* group, *Out-Module* group, and *Constant* group) by the function relevance to make sure the parallel CPU cores run during multiple time step simulation. *In-Module* variables are the variables modified by a specific module and can be directly retrieved from the end of the module. These variables appeared in the outStream data flow of the module to drive the subsequent module. *Constant* variables are the environmental setting variables whose values are fixed at the beginning of the model running; hence, they only need to be initialized at the very first time step in a multi-time step simulation. *Out-Module* variables are the input variables whose values are modified by other modules. As such, we need to retrieve them from other modules during the original scientific code running and then provide them at each time step in the unit test platform. By analyzing how these variables were used, we further tagged them with two tags: disk variables and memory variables. The *Constant* variables refer to variables whose values keep constant during the experiment process, we tag them as disk variables and only read them once. The *Out-Module* variables refer to the ones whose values are received from outside of the target modules; we tagged them as disk variables and read them at each time step. Since the *In-Module* variables refer to variables whose values are changed during the experimental process, we tagged them as memory variables and transfer them to the next time step.

Third, we designed a loop-parallel algorithm for an an n-case computation illustrated in Figure 1. First, the retrieved *Constant* variables were used to set up the experiment environment. Then, n cases were initialized with a customized requirement and MPI execution environment. Inside each case, there were *rank-size* processes. Every process i read data from two storage media (except the first timestep process loads all variables from disk). One storage media was disk, which was a file written with *Out-Module* variables from the original scientific code. The other storage media was MPI message buffer, which received updated *In-Module* data from the processes at the previous timestep. The process simulated the status of timestep T ranging from 1 to user-defined F and constantly sent data to the next timestep computation on the No. *(i+1) mod ranksize* process.

4 Implementation

The development platform used in the study is a multi-programmatic heterogeneous federated cluster with the Red Hat Enterprise Linux (RHEL) operating system. The production unit testing platform runs on Titan which contains 18,688 physical compute nodes, each with a processor, physical memory, and a connection to the Cray custom high-speed interconnect. Each compute node houses one 16-core 2.2GHz AMD Opteron^{TM} 6274 (Interlagos) processor and 32 GB of RAM. The working procedure of the parallel unit testing framework (PUTF, as shown in Figure 2) has five stages: user specification, dataflow generation, customized experiment generation, experiment verification, and experiment execution.

User Specification: The first step is to define which module to isolate. Generally, in the model build step, the PUTF claims which modules to extract based on user requirments; the researchers customize their experiment by designing the necessary duration of the experiment simulation period and providing initial parameters. Dataflow Generation: PUTF uses a dataflow analyzer to split data dependency between modules. The analyzer first collects constant variables, in-module variables, and out-Module variables inside the code, and then inserts all

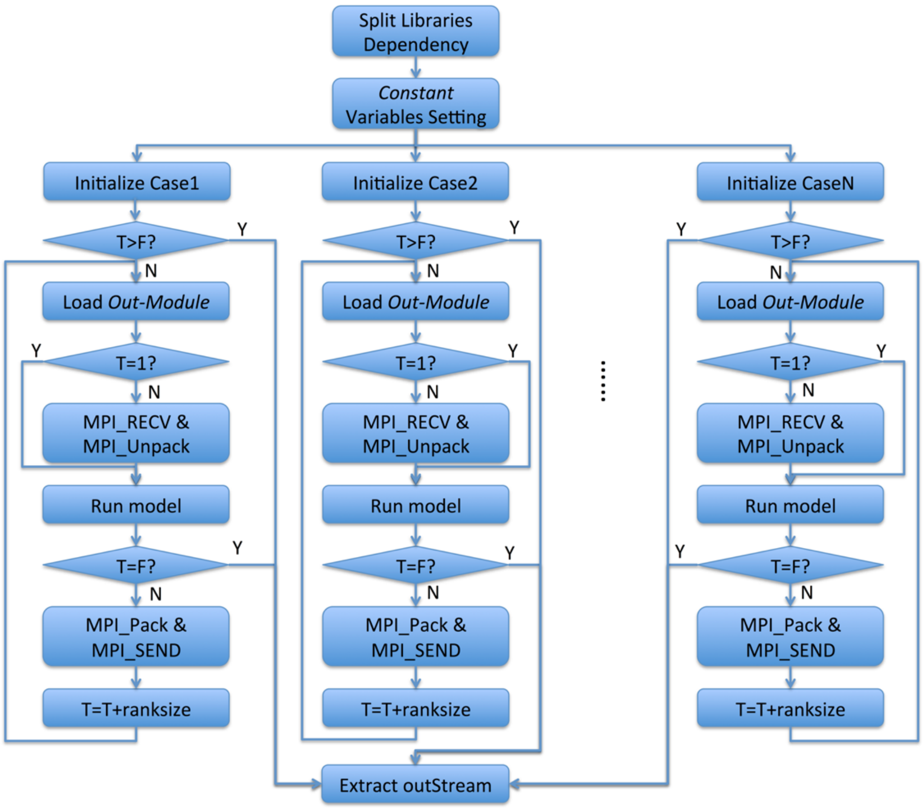


Fig. 1: Loop Parallel method

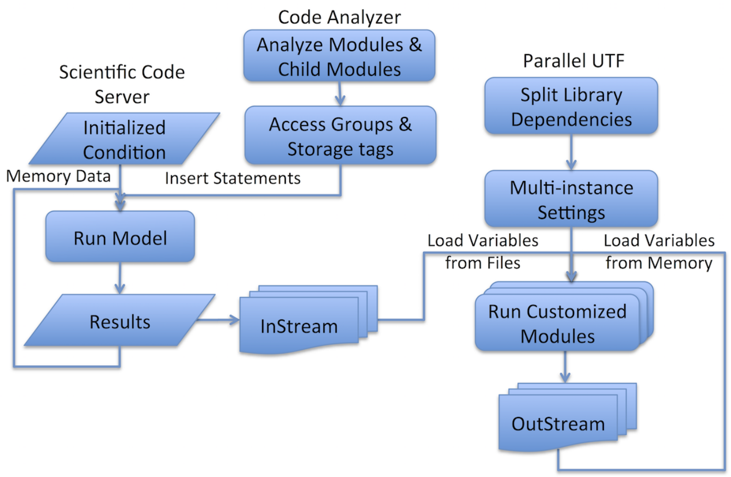


Fig. 2: Overview of the Improved Parallel Infrastructure

these variables declarations to the corresponding modules in the original code as an "inspector". After re-compiling and re-running the scientific code, we can extracted all required input data stream files and starting timestep output data stream file. The starting timestep output data stream file is used to verify the logic of the PUTF. A data generation script scans all user-specified modules using the dataflow analyzer, then collect, divide, and extracts data stream for module-based simulations.

Customized experiment generation: In this stage, it is necessary to isolate modules to be independent of other unnecessary libraries, such as the parallel IO library (PIO) and Networked Common Data Format (netCDF), which is complicated with platform incompatibly problems. Second, these libraries were replaced with easily implemented functions without dependent libraries, making the kernel more portable. Finally, a driver is configured with initial global parameters and constant variables. At last, the PUTF prepares the required modules and loads required data from different storage media based on the recursive analysis mentioned in the previous step.

Experiment Verification: in order to verify that each module works correctly on our platform with the previous setting, we compare results from the unit testing platform with results from the scientific code. At this step, the environmental setting and parameter initialization are the same as the original scientific code. This step is tested quickly using one timestep running.

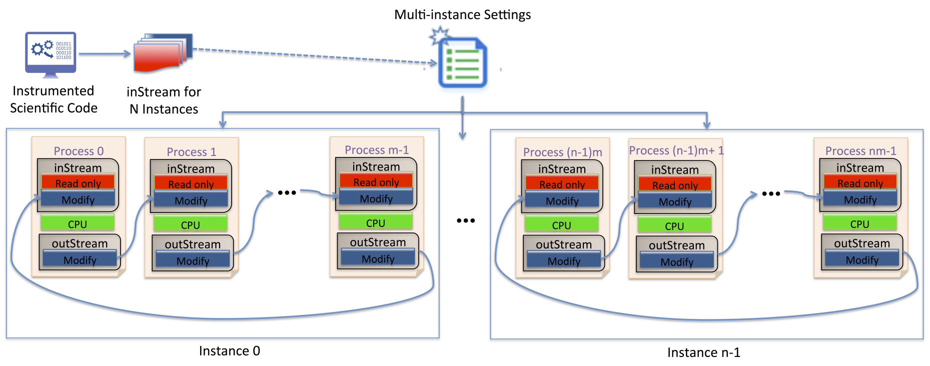


Fig. 3: Overview of MPI Unit platform

Experiment Execution: Once the infrastructure settings are verified, we run the experiment in parallel. Figure 3 shows how the parallel framework works. At the beginning of the experiment, we apply $n*m$ processes for n instances. Inside every instance, a process first loaded *Out-Module* variables from a disk file and then checked whether it is the first time step. If the process is not the first timestep verification, it waits *In-Module* data package through *MPI_RECV* method from the previous timestep. Otherwise, the process read disk file for initialization. A very short time later, the process finishes the computation and checks if the process is at the last timestep, so that it can record the experiment result and exit; otherwise, it sends the *In-Module* data package to process at the next timestep through *MPI_SEND* and begins to deal with calulations at the No. *i+m* timestep. Different instances shares the same disk files but conducts different computation.

5 Case Study

5.1 Scientific Code and Module-based Experiments

The "Accelerated Climate Model for Energy (ACME)" a fully-coupled Earth system model development and simulation project to investigate energy-relevant science using code optimized for advanced computers. Inside ACME system, the ACME Land model (ALM) is designed to understand how natural and human changes in terrestrial land surfaces will affect the climate [5]. The ALM model consists of submodels related to land biogeophysics, the hydrologic cycle, biogeochemistry, human dimensions, and ecosystem dynamics. Due to internal biogeophysics and geochemical connections, ALM simulations have to be executed with other earth system components within ACME, such as atmosphere, ocean, ice, and glaciers etc. [8].

The objective of this case study was to compare the performance of the decomposition reaction network within the ALM using data collected from the long-term intersite decomposition experiment team (LIDET). However, with more than 1800 source files and over 350,000 lines of source code, the software complexity of ALM became a barrier to rapid model improvements and validation [9] [10], also it is very inconvenient to track specific modules and capture the impact of specific factors on the overall model performance.

At the center of ALM decomposition submodel is the Convergent Trophic Cascade (CTC) method. We would like to evaluate CTC using LIDET data. In a previous study [1], CTC was investigate in standalone mode without consideration of temporal variations in environmental and nutrient conditions that would occur in the full model. If we want to perform the LIDET study in ALM model directly, that may introduce unrealistic feedbacks between the simulated litter bags and vegetation growth. Therefore, we develop a model within our PUTF framework which allows the CTC submodel to operate independently while retaining the temporally varying environmental drivers calculated by ALM. Particularly, we are interested in (1) the influence of litter decomposition base rate parameters, and (2) the influence of nitrogen limitation, and the temporal variability of this limitation, on litter decomposition.

5.2 Experiment setups

In the experiments, six types of leaf litter were placed in fine mesh bags at 21 sites representing a range of biomes. The mass of remaining carbon and nitrogen in this litter was measured annually over a 10-year period. To simulate the experimental conditions in the model, we first spined up the carbon and nitrogen pools using an accelerated decomposition phase for 500 years, followed by a return to normal decomposition rate for 500 years [6]. In these simulations, we used a repeating cycle of the CRU-NCEP meteorology over the years 1901-1920. Then we performed a transient simulation from the years January 1st, 1850 - October 1st, 1990, which was forced by changing atmospheric CO2 concentrations, nitrogen and aerosol deposition and land use. Globally gridded meteorological and land-surface data were used for these simulations except for plant functional type, which was replaced using site information. The model state on October 1st, 1990 represents the simulated conditions at the beginning of the experiment.

At this point, the UTF framework is used to execute a 10-year simulation. For a control simulation in which no litter is added, we run the full scientific model and save all model state variables at every timestep. These model states were then used as boundary conditions for our decomposition unit, for which only the decomposition subroutines and relevant updating codes are active. For each site, we added litter inputs to the first soil layer using the appropriate mass, quality, and C:N ratios for each of the six litter types. The decomposition unit is driven by soil moisture, temperature, and the nutrient limitation factor for decomposition from the full model. Unlike in the full version of the scientific code, there was no feedback between decomposition and the ecosystem.

5.3 Results and Analysis

In this section, we used a dynamic performance analysis measurement tool to help to improve the framework.

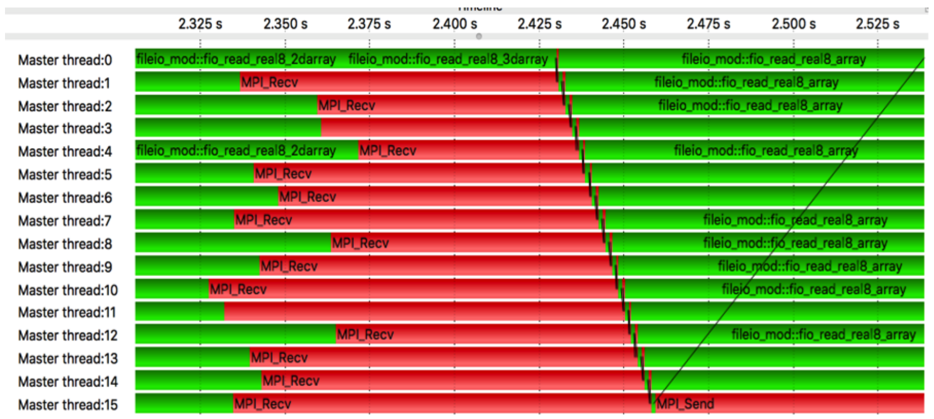
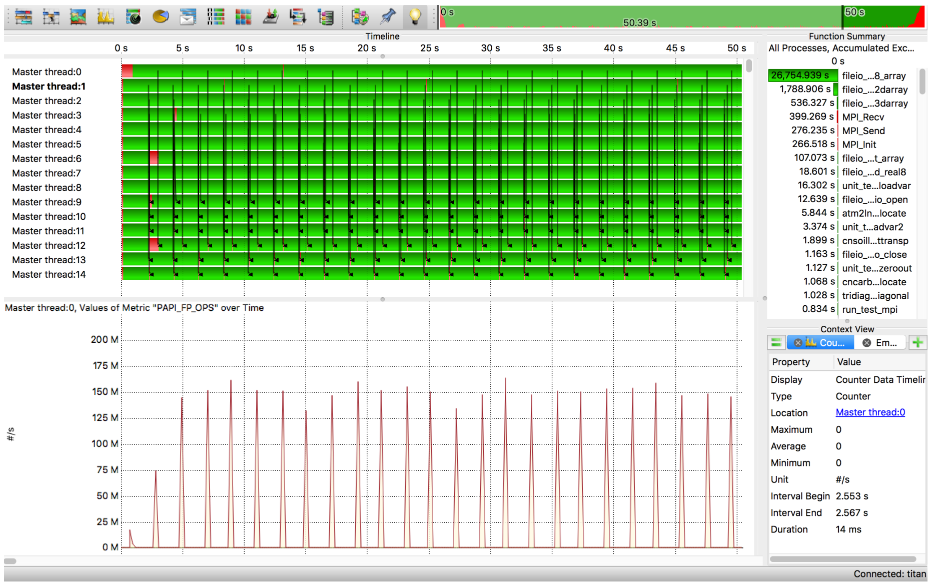


Fig. 4: Timeline chart

5.3.1 Parallel I/O In this experiment, if we applied up to 2114 processes and 133 computing nodes in Titan, the total execution time for one site and one leaf liter's type in 4-month's simulation was 13s, the best performance among configurations. In Figure 4, we applied 16 processes and 1 node to simulate one site and one type's 10 years's simulation. The red bar signifies the MPI functions, which include *MPI_Init*, *MPI_Recv*, *MPI_Send* and *MPI_Finalize*. The green bar between red ones represents CPU computation, the green bars after *MPI_Send* and before *MPI_Receive* are disk I/O read. Messages exchanged between different processes are depicted as black lines. Within the case, since every time step needs previous time step's data as input, the function computation is sequential, while the I/O operation is parallel. The *MPI_Recv* bar is long over time because every process is waiting for previous results.



Function Summary

All Processes, Accumulated Exc...

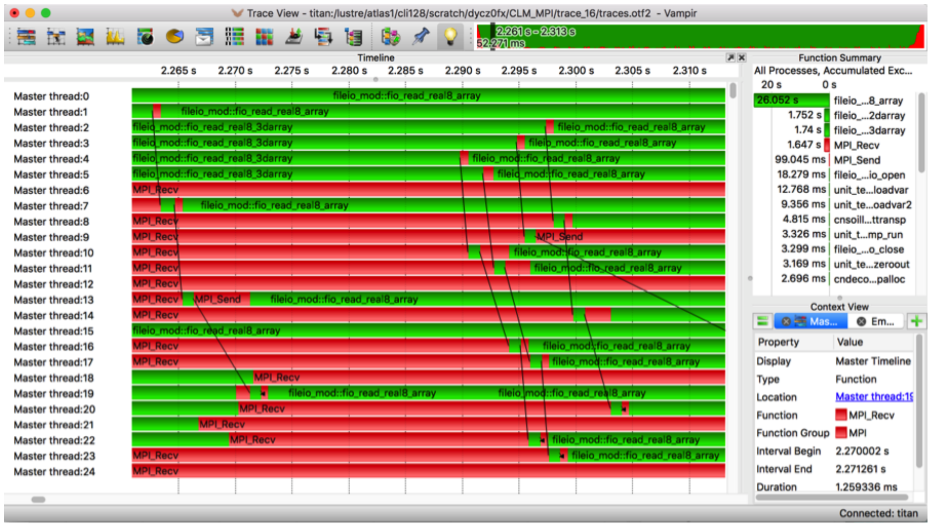
0 s

26,754.939 s	fileio_...8_array
1,788.906 s	fileio_...2darray
536.327 s	fileio_...3darray
399.269 s	MPI_Recv
276.235 s	MPI_Send
266.518 s	MPI_Init
107.073 s	fileio_...t_array
18.601 s	fileio_...d_real8
16.302 s	unit_te...loadvar
12.639 s	fileio_...io_open
5.844 s	atm2ln...locate
3.374 s	unit_t...advar2
1.899 s	cnsoill...ttransp
1.163 s	fileio_...o_close
1.127 s	unit_te...zeroout
1.068 s	cncarb...locate
1.028 s	tridiag...iagonal
0.834 s	run_test_mpi

Property	Value
Display	Counter Data Timelir
Type	Counter
Location	Master thread:0
Maximum	0
Average	0
Minimum	0
Unit	#/s
Interval Begin	2.553 s
Interval End	2.567 s
Duration	14 ms

Fig. 5: Profiling Info for improved PUTF

5.3.2 MPI with Parallel I/O In Figure 5 and Figure 6, we applied 600 processes and 40 nodes to parallel one site and 6 types 10-years simulation experiment. The red box stands for the MPI functions, and the green boxes consist of function computation and I/O operation. The black line represents the paralleled CPU and how the MPI message goes. In this case, every type in the same site shared the same input data but computed in different ways. Therefore the function execution and the I/O operation were both parallel which improved the overall performance that can be seen from the counter in Figure 5 and Figure 6.



26.052 s	fileio_...8_array
1.752 s	fileio_...2darray
1.74 s	fileio_...3darray
1.647 s	MPI_Recv
99.045 ms	MPI_Send
18.279 ms	fileio_...io_open
12.768 ms	unit_te...loadvar
9.356 ms	unit_te...oadvar2
4.815 ms	cnsoill...ttransp
3.326 ms	unit_t...mp_run
3.299 ms	fileio_...o_close
3.169 ms	unit_te...zeroout
2.696 ms	cndeco...palloc

Property	Value
Display	Master Timeline
Type	Function
Location	Master thread:1
Function	🟥 MPI_Recv
Function Group	🟥 MPI
Interval Begin	2.270002 s
Interval End	2.271261 s
Duration	1.259336 ms

Fig. 6: Improved Time dissection of CPU and I/O

5.4 Experiment Results

We compare the full version of ALM with ALM UTF for conifer and tropical forests against LIDET observations, representing 5 and 4 of the 21 sites respectively. Figure 7 shows the remaining mass of carbon as a function of time over the 10-year experiment for the two model versions and observations, averaged over all 6 litter types. A best fit to observations is performed by fitting an exponential function y = a*exp(-bx) + c. In both conifer and tropical forests, the carbon mass remaining declines more rapidly in ALM UTF than in the full ALM, which is more consistent with the best fit to observations. The ALM UTF model is more consistent with the actual experimental conditions, because in the experiment the small amount of litter in the litter bag added to each plot is not large enough to induce ecosystem-scale feedbacks. However in the full ALM, the added litter effectively covers the entire land surface, causing feedbacks to vegetation growth. The additional litter unrealistically stimulates vegetation growth in the full ALM, causing more carbon fixation and increased litterfall, effectively increasing the carbon mass remaining. In both the full ALM and in the ALM UTF, the carbon mass remaining is significantly higher than that estimated by [1] when comparing the CTC submodel in ALM to the DAYCENT soil decomposition model. That analysis, while also using a functional unit approach, did not use the full model for boundary conditions and thus neglected to consider changing nutrient limitation and environmental conditions. The approach taken in ALM UTF is a useful way to perform such model-experiment intercomparisons in a consistent way while avoiding unrealistic feedback in small-scale experiments.

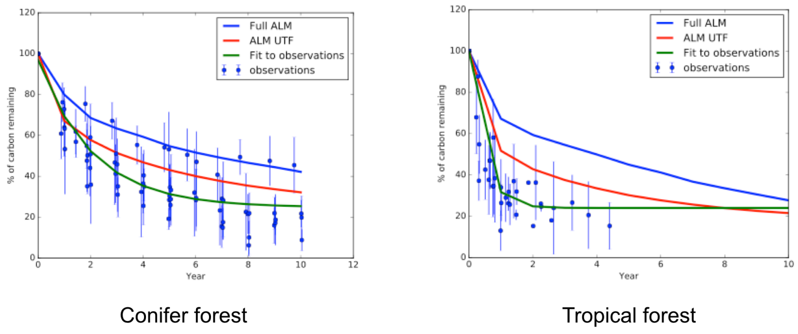


Fig. 7: Comparison among full version of ALM, ALM UTF and LIDET observations. Average carbon mass remaining in relation to time for leaf litter decomposed at two sites. Data are averaged across six leaf litter types for sites classified as conifer forest and tropical forest

6 Conclusions

Large-scale scientific code is important for scientific research. However, because of the complexity of models, it is very time-consuming to modify scientific code and to validate individual modules inside a complex modeling system. To facilitate module evaluation and validation within scientific applications, we first introduce a unit testing framework. Since scientific experiment analysis generally requires a long-term simulation with large-amount of data, we apply message passing based parallelization and I/O behavior optimization to improve the performance of the unit testing framework on parallel computing infrastructure. We also used profiling result to guide the parallel process implementation. Finally, we use a standalone moduled-based simulation, extract from a large-scale Earth System Model, to demonstrates the scalability, portability, and high-efficiency of the parallel functional unit testing framework.

Acknowledgement

Some part of this research is included in Yao's Ph.D. dissertation (A Kernel Generation Framework for Scientific Legacy Code [12]) with the University of Tennessee, Knoxville. TN. This research was funded by the U.S. Department of Energy, Office of Science, Biological and Environmental Research program (E3SM and TES) and Advanced Scientific Computing Research program. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. Bonan, G.B., Hartman, M.D., Parton, W.J., Wieder, W.R.: Evaluating litter decomposition in earth system models with long-term litterbag experiments: an example using the community land model version 4 (clm4). Global change biology **19**(3), 957–974 (2013)
2. Brunst, H.: Integrative concepts for scalable distributed performance analysis and visualization of parallel programs. Shaker (2008)
3. Kim, Y., Dennis, J., Kerr, C., Kumar, R.R.P., Simha, A., Baker, A., Mickelson, S.: Kgen: A python tool for automated fortran kernel generation and verification. Procedia Computer Science **80**, 1450–1460 (2016)
4. Loh, E.: The ideal hpc programming language. Communications of the ACM **53**(7), 42–47 (2010)
5. Oleson, K.W., Lawrence, D.M., Gordon, B., Flanner, M.G., Kluzek, E., Peter, J., Levis, S., Swenson, S.C., Thornton, E., Feddema, J., et al.: Technical description of version 4.0 of the community land model (clm) (2010). https://doi.org/10.5065/D6FB50WZ
6. Thornton, P.E., Rosenbloom, N.A.: Ecosystem model spin-up: Estimating steady state conditions in a coupled terrestrial carbon and nitrogen cycle model. Ecological Modelling **189**(1), 25–48 (2005)
7. Vardi, M.: Science has only two legs **53**, 5 (09 2010)
8. Wang, D., Janjusic, T., Iversen, C., Thornton, P., Karssovski, M., Wu, W., Xu, Y.: A scientific function test framework for modular environmental model development: application to the community land model. In: Proceedings of the 2015 International Workshop on Software Engineering for High Performance Computing in Science. pp. 16–23. IEEE Press (2015)
9. Wang, D., Post, W.M., Wilson, B.E.: Climate change modeling: Computational opportunities and challenges. Computing in Science & Engineering **13**(5), 36–42 (2011)
10. Wang, D., Schuchart, J., Janjusic, T., Winkler, F., Xu, Y., Kartsaklis, C.: Toward better understanding of the community land model within the earth system modeling framework. Procedia Computer Science **29**, 1515–1524 (2014)
11. Wang, D., Yuan, F., Hernandez, B., Pei, Y., Yao, C., Steed, C.: Virtual observation system for earth system model: An application to acme land model simulations. International Journal of Advanced Computer Science and Applications **8**(2) (2017). https://doi.org/10.14569/IJACSA.2017.080223, http://dx.doi.org/10.14569/IJACSA.2017.080223
12. Yao, Z.: A Kernel Generation Framework for Scientific Legacy Code. Ph.D. thesis, University of Tennessee, Knoxville (2018)
13. Yao, Z., Jia, Y., Wang, D., Steed, C., Atchley, S.: In situ data infrastructure for scientific unit testing platform. Procedia Computer Science **80**, 587–598 (2016)