Journal of Software Engineering and Applications, 2016, 9, 199-207
Published Online May 2016 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2016.95016

Scientific Research Publishing

Parallelization of Diagnostics for Climate Model Development

Jim McEnerney[1], Sasha Ames[1], Cameron Christensen[2], Charles Doutriaux[1], Tony Hoang[1], Jeff Painter[1], Brian Smith[3], Zeshawn Shaheen[1], Dean Williams[1]

[1]Lawrence Livermore National Laboratory, Livermore, California, USA
[2]University of Utah, Salt Lake City, Utah, USA
[3]Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
Email: mcenerney1@llnl.gov, ames4@llnl.gov, doutriaux1@llnl.gov, hoang1@llnl.gov, painter1@llnl.gov, shaheen2@llnl.gov, williams13@llnl.gov, scicameron@gmail.com, smithbe@ornl.gov

Received 26 March 2016; accepted 21 May 2016; published 24 May 2016

Abstract

The parallelization of the diagnostics for climate research has been an important goal in the performance testing and improvement of the diagnostics for the Department of Energy's (DOE's) Accelerated Climate Modeling for Energy (ACME) project [1]. The primary mission of the ACME project is to build and test the next-generation Earth system model for current and future generations of computing systems operated by the DOE office of science computing facilities, including the envisioned exascale systems foreseen in the early part of the next decade. As part of the underpinning workflow environment, a diagnostics, model metrics, and intercomparison Python framework, called UVC Metrics was created to aid in testing and production execution of the model. This framework builds on common methods and similar metrics to accommodate and diagnose individual component models, such as atmosphere, land, ocean, sea ice, and land ice. This paper reports on initial parallelization of UVC Metrics for the atmosphere model component using two popular frameworks: MPI and SPARK. A timing study is presented to assess the performance of each method in which significant improvement was achieved for both frameworks despite I/O contentions with NFS. The advantages and disadvantages of each framework are also presented.

Keywords

Climate Diagnostics, Parallel, MPI, SPARK

1. Introduction

Research into climate change has many computational requirements starting with the execution of multi-physics

How to cite this paper: McEnerney, J., Ames, S., Christensen, C., Doutriaux, C., Hoang, T., Painter, J., Smith, B., Shaheen, Z. and Williams, D. (2016) Parallelization of Diagnostics for Climate Model Development. *Journal of Software Engineering and Applications*, **9**, 199-207. http://dx.doi.org/10.4236/jsea.2016.95016

climate simulations [1]. Support for this project has evolved into a computing ecosystem, in the sense that all aspects of data, software, libraries and tools, network and transfer mechanisms, storage, as well as requirements for computing hardware are included. These have the potential impact for advancing the scientific mission of climate science. Data management has grown in size and geographical requirements that literally span the globe [2].

There are several opportunities to parallelize applications that support climate research. While the phrase "embarrassingly parallel" is frequently used, it does not reflect the amount of effort required to succeed in the conversion of an application from serial to parallel. When confronted with such a project, there are important basic questions to answer. Is there some part of the algorithm that is parallelizable? Does it require reimplementation of any kind? What framework should be chosen for implementation? What performance increase should be expected? In the context of climate data a good application involves the reduction of a large quantity of data. At what point does saturation set in, beyond which no possible improvement can be gained, and perhaps, is there a performance degradation?

While performance is the overarching goal, any implementation cannot ignore the data requirements. Specifically, terabytes and petabytes of data cannot be moved to local machines; remote processing for analysis and data reduction is imperative. This will only get worse with either higher resolution simulations as in the ACME project [1] or more long-term simulations as in CMIP [3] [4]. It is natural to reverse this thinking and push computations to the data, potentially across different geographical computing facilities [5]. Furthermore, due to the length of time to complete a simulation, it is a requirement to allow for "*in situ* analysis", that is, performing analysis while the model is running.

Constraints are often imposed with such requirements. For example, the computer architecture is already defined with mainstream technologies, such as HPC, cloud clusters, linux clusters, GPUs, and CPUs. There are also software constraints starting with languages. Python is the chosen language in this case and already there is a performance loss with an interpreter. While Python may not be the best language for parallelization, it does maintain the balance between performance and software maintainability. Also, Python, as does Scala, has a benefit in supporting functional language features where these features facilitate runtime scheduling of computation kernels when defined within a function. The performance overhead would be negligible or latency hidden by I/O. While programming in a language compiled to traditional machine code may lack these overheads, much time would be spent blocking on I/O. Further constraints include compilers and libraries available on these systems, in addition to their versions.

The intent of this paper is to provide insight into answers to these questions as applied to the component model diagnostics. To our knowledge, no effort has previously been made to evaluate parallel climate model output diagnostics, as presented here.

2. Climate Diagnostics

The climate diagnostics are available in the Climate Data Analysis Tools (CDAT) framework. CDAT was developed at Lawrence Livermore National Laboratory (LLNL) for the analysis, visualization, and management of large-scale distributed climate data [6]-[8]. This toolkit is implemented in Python and incorporates many packages including numpy, vcs, and others.

The atmospheric diagnostics package provides the climatologist a means to assess climate simulations relative to experimental data or another simulation. They are available from the command line or the UV-CDAT GUI. It is common to generate a graphical image in the form of a Mercator projection with the relevant parameters under consideration. An alternate two-dimensional image showing latitude versus month of the year is displayed in **Figure 1**. This diagnostic is used in the study.

Climate data are frequently 4 dimensional: latitude, longitude, altitude (level) and time. As a simulation executes, these data are recorded in history files for subsequent processing or a restart capability. Climatologies are averages over some number of coordinates listed above. Parallelizing the averaging process is our primary focus for parallelization in the climate project.

The design of the diagnostics system for the ACME project is based on a central class that defines or stages the computation to be performed. All of the associated attributes including the name of the function used for the reduction and information for graphical output are defined. The actual computation is not performed during the initialization but later with a call to the objects results method. Once completed, graphical output is generated.

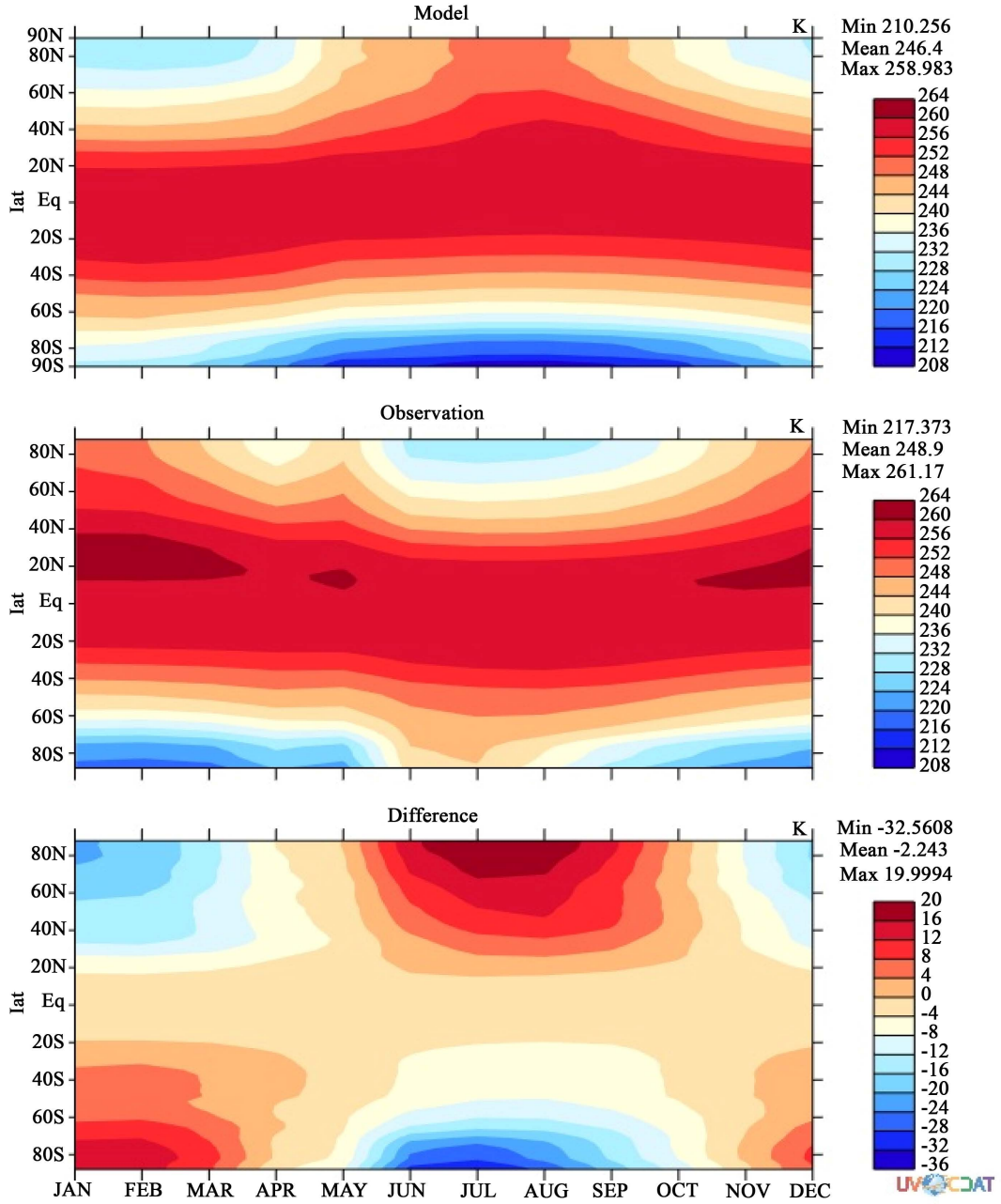

Figure 1. Annual cycle contour plot of zonal mean.

This structure of diagnostics enables the use of parallelization since each process can be assigned a different reduction.

There are several types of reductions available. Each one is a function that reduces the data by averaging across some of the four axes and reduces to fewer axes for display purposes. The type of reduction depends on what is being studied and is specified in the object. Climate simulations produce significant amounts of data located in several files and the reduction process span these files. Thus the current implementation of diagnostics has file I/O interleaved with the computation.

3. The Performance Test

This project was an initial parallelization project intended to penetrate the implementation of one diagnostic and

parallelize at more than a superficial level for the computation of climatologies. There are deeper, more difficult levels of the implementation where even more performance could be achieved but are outside the scope of this study. The diagnostic chosen is the Annual Cycle Contour Plot of Zonal means. The graphical output is a plot of latitude versus month, **Figure 1**. The reduction averages across longitude and altitude.

There are 24 independent calculations: 12 models and 12 observations. The point where the parallelization occurs is where a dictionary is constructed with all of the 24 reduced variables defined and the reduction process begins. For MPI the scatter/gather pair is used. Once implemented for MPI, the SPARK implementation was straightforward for this application; this same loop was simply mapped and the entries in the dictionary were reduced.

Data from CMIP5 was used for this study. CMIP5 is one project in a series aimed at multi-physics climate simulations to understand both near term (out to about 2035) and long term (out to 2100 and beyond) predictions [3]. The data used for the model are 17 GB. A separate copy of these data was used for the observation. This was done to bring the total amount of data to a reasonably high level. Note, while observational data is frequently much smaller, there are future requirements for high resolution data. The simulations span the years from 2000 to 2670 for each month with a total of 67 files, one for each decade. The individual arrays have dimensions (time, level, latitude, longitude) with size (120, 17, 128, 256); 120 covers 10 years of simulated data.

The test results reflect the time required for the loop over of the reduced variable dictionary, not the timing of the entire application. The test ensured that each core was processing the same number of months, that is, the same amount of data. For this study the 24 independent calculations enabled several different configurations. For MPI, the number of nodes and number of tasks per node are specified. For SPARK, the number of partitions is specified. For the purpose of comparison the number of partitions is simply the product of node number and task number. Each execution was repeated a nominal 5 times to identify inconsistencies and to compute means and standard deviation. A weak scaling test was also performed, wherein the workload started with 3 months and increased linearly as the number of nodes increased.

4. Performance Configuration Setup

The hardware configuration used in the study is a Linux cluster. The master node has 4 Quad core 2.6 GHz Intel Xeon E5-2670 CPUs and 256 GB RAM with hyper-threading enabled. The worker nodes have 2 8-Quad core 2.4 GHz Intel Xeon E5-2650 CPUs and 128 GB RAM, running on a private 10 Gigabit-Ethernet network, with CentOS Linux 6.x, and hyper-threading disabled. The master node has an NFS drive running version 4.x. This performance study is executed as the only user of these cluster nodes; the NFS mount is accessed in a shared environment on other servers, which is standard practice in most computing environments.

MPI is an established standard for parallel computing that has origins dating to the early 1990s, [9]-[13]. The python package is mpi4py version 1.3 [14]-[16]. There are many options that can be specified with MPI. The two most relevant used are number of nodes and number of tasks per node. Job management was accomplished with SLURM. It is a resource manager that was first developed at LLNL and provides the basics needed to run a parallel job, namely, resource allocation, job executing/monitoring and queue managing of pending jobs, [17]-[19]. When coupled with MPI it provides a complete environment for parallel computing.

SPARK is an implementation of the map/reduce paradigm [20] and boasts impressive performance improvements over Hadoop. At some level SPARK incorporates both the functionality of MPI and SLURM. Although it makes the architecture of the cluster more opaque, it requires only the number of partitions or cores to be used. Each node has an executor that manages the node. One advantage of this framework is that jobs far too massive for the resources will be broken down into “bite size” chunks and executed in out-of-core fashion. You can make the chunks too large for a single worker process, but having more workers wouldn’t make a difference. For the study version 1.5.2 was used.

5. Timing Results

Timing for the serial calculation was 1902 seconds. For MPI, simply increasing the number of nodes does not necessarily improve performance, **Figure 2** and **Table 1**. With the increase from 3 to 4 nodes and a single task per node, the performance inexplicably degrades by about 10%. This decline appears to be some sort of contention accessing the NFS drive. However, as the number of nodes continues to increase the performance does also, although it appears to be only a marginal increase with 2 tasks per node. Also comparing 1 task per node versus 2 tasks per node, there is mostly a uniform improvement.

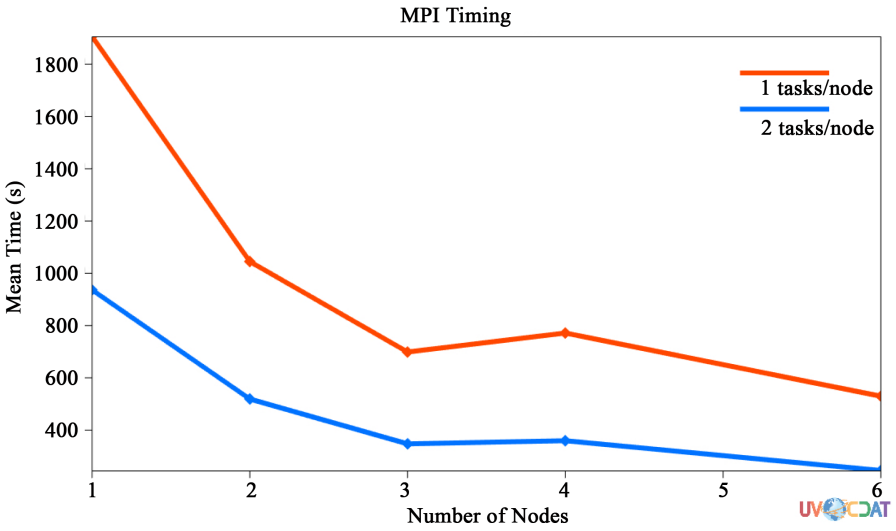

Figure 2. MPI mean time vs. number of nodes.

Table 1. MPI timing table.

#nodes	#tasks	mean	std
1	1	1904.8	6.58
1	2	935.04	0.53
2	1	1043.41	1.33
2	2	517.57	1.1
3	1	697.38	1.56
3	2	346.02	0.73
4	1	769.99	27.33
4	2	358.01	22.75
6	1	528.25	27.88
6	2	244.06	23.84

The SPARK timing is displayed in **Figure 3** and **Table 2** as two separate curves. The red curve is the result of restricting the configuration on the worker nodes so that only a single process operates at a time. This was done for a comparison with MPI to be discussed shortly. The configuration for the blue curve was unrestricted but only two tasks per node actually executed. Note the following specifics of the measurement: the x-axis represents the number of partitions running, rather than the number of nodes (which may each have one or more tasks) as shown in **Figure 2**; this setup is only valid for this test and it would not be used in practice. Nonetheless, our observations give an indication of what should be the best way of configuring SPARK for this application.

The second configuration with more than one task per node gives the best performance, which improves steadily as the number of partitions increases, although it too seems to be close to steady state with 3 or more partitions. There is anomalous timing in the 4 node/1 task versus 2 nodes/2 tasks. The standard deviations are quite large and is interpreted as an NFS issue as in the case of MPI.

The only fair comparison between MPI and SPARK is to consider the case of a single task per node in the case of MPI and the number of nodes is treated as the number of partitions for SPARK. This contrived setting is only valid for a performance test and it is invalid in an operational situation. In **Figure 4** it is evident that the two are comparable. SPARKs performance in the best case is very similar to MPIs, seen by comparing 12 parti-

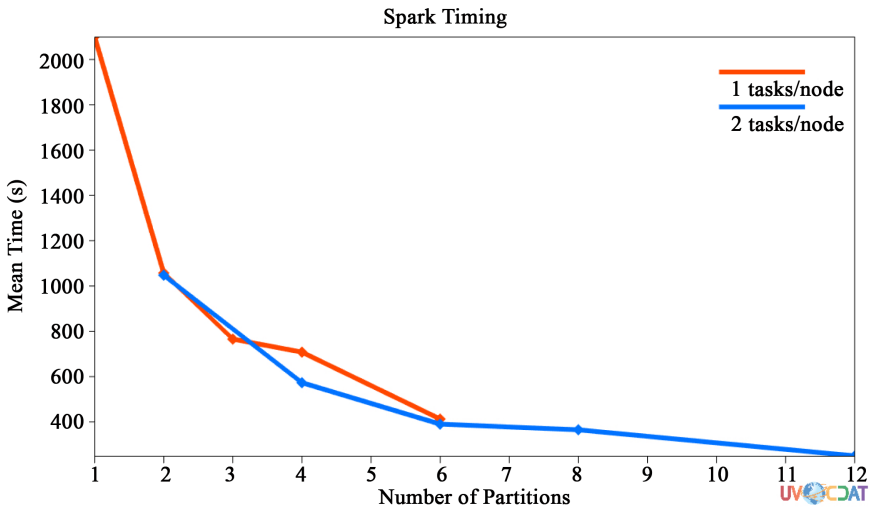

Figure 3. SPARK mean time vs. partition count.

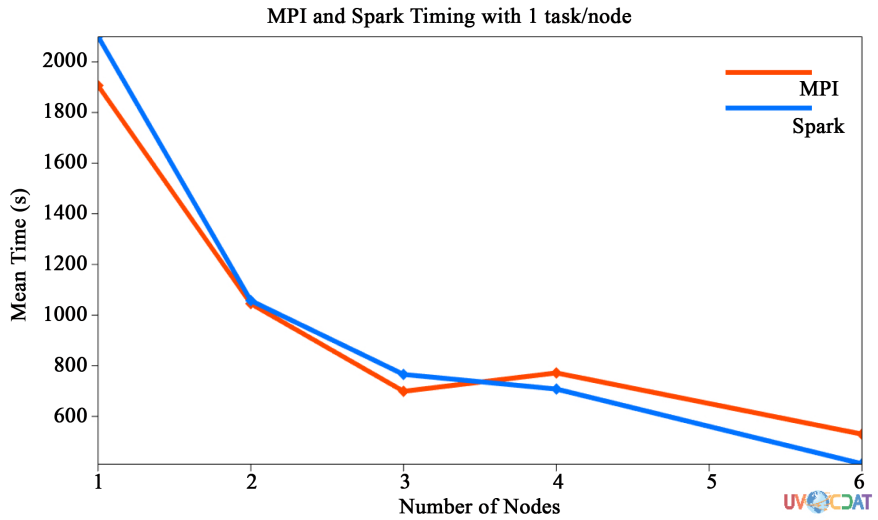

Figure 4. MPI/SPARK timing comparison.

Table 2. SPARK timing table.

#nodes	#tasks	#partitions	mean	std
1	1	1	2099.05	11.6
1	2	2	1045.92	2.49
2	1	2	1055.62	1.22
2	2	4	571.6	76.25
3	1	3	764.0	107.81
3	2	6	388.61	50.32
4	1	4	706.35	153.15
4	2	8	363.52	45.57
6	1	6	411.19	22.25
6	2	12	248.4	15.69

tions for SPARK and 6 nodes/2 tasks for MPI. Note in this case there is a little more consistency for SPARK over MPI; the standard deviations are different.

To study the weak scaling, the amount of work for each processor was linearly increased starting with 3 months for a single processor up to 18 months for 6 processors. By doing this only the length of computation was being measured, not all of the 24 months were computed. Again 5 executions were performed. It would be unexpected to see the amount of time to perform the computation to be constant, the perfect expectation. A comparison indicates that SPARK is more consistent, **Figure 5**. For MPI there is a noticeable degradation as the node count increases from 3 to 4. Note this anomaly appears in the MPI timing, see **Figure 2**.

6. Experience with Parallel Frameworks

Troubleshooting in either framework reflects a significant issue found in any parallel framework. The methods used for serial applications to troubleshoot provide only the basics for parallel applications. Both frameworks have many moving parts and diagnosing issues is difficult. For example, when a node is down it is difficult to determine where the problem is and there is little evidence for troubleshooting. SLURM provides rudimentary diagnostics for system status. SPARK's web interface for running applications is quite good, which is much better than MPI. If you need to use a specific Python the PYSPARK_PYTHON environment works fine. However, if the workers on each node need to use this same Python, a typical situation, there is a configuration issue. This does not fit a multi user environment. In both cases there was a limitation on the number of jobs that can run simultaneously on this cluster; 2 was the maximum. There was no real indication what the underlying reason was. In fact, it may be associated with one of the packages used by UVCDAT.

MPI has a high level of configurability, which translates into more complexity for a developer. One observation apparent from the graphs is that it requires experimentation to see what is the best configuration of number of nodes and the number of tasks per node. It seems that the developer must know the architecture of the network and the same holds for a user, to a lesser extent.

SPARK works best with a well-defined encapsulated computation. For example, perform file I/O to stage data in the mapper and the actual computation in the reducer. There are several configuration parameters available to the application from each framework. Some experimentation is required to identify the best setting. With SPARK, the output from workers is not integrated with the master output, although there are techniques for doing this.

7. Conclusions

Data ingesting with NFS is a known problem. Too many processes accessing data cause bottlenecks in these systems and using centralized disk storage are definitely a problem. This showed up with as few as 12 processes in either MPI or SPARK. Using a file system other than NFS such as HDFS, Tachyon (with SPARK), or a

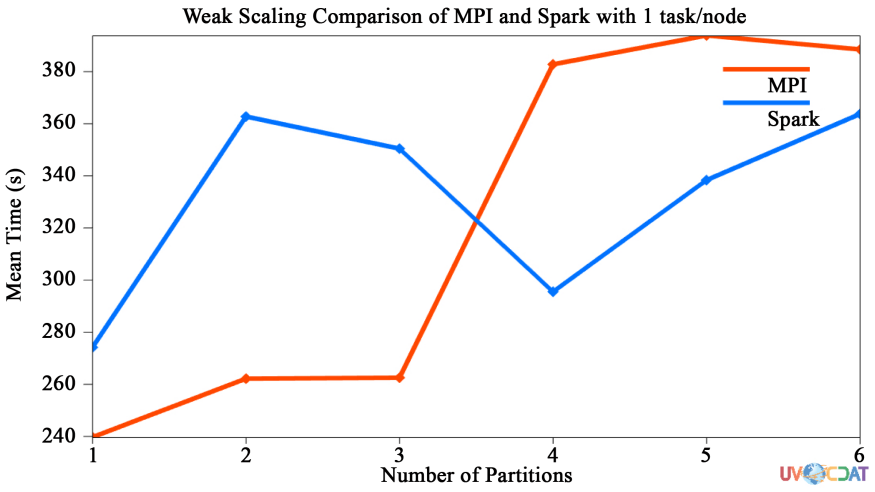

Figure 5. MPI/SPARK weak scaling comparison.

bona fide parallel file system (using MPI-IO) such as Lustre or GPFS should be considered to improve the situation. If successful this may be part of a justification for a move to other technologies. Several authors have had similar experience with NFS limitations in comparable parallel access environments [6].

A more granular computation can be employed to increase parallelization even further. Specifically, push the parallelization to a deeper level such in the libraries. For example the CDMS library, which provides all of the functionality to manipulate climate data arrays, is a likely target. Internal memory analysis would be needed to identify how to achieve the best performance.

Both frameworks have a broad development community. A developer new to parallel programming should consider SPARK. It's more user-friendly than MPI, although the newness of SPARK places the burden of administration on the developer. SPARK provides arbitrary scalability without the user needing to write special out-of-core data processing code as would be required for MPI. This is a distinct advantage available to novice users that would be difficult to match even by a seasoned MPI developer.

Acknowledgements

Much appreciation goes to Angela Jefferson for her editorial skills. This work was supported by the U.S. Department of Energy Office of Science/Office of Biological and Environmental Research under Contract DE-AC52-07NA27344 at Lawrence Livermore National Laboratory.

References

[1] ACME Council (2014) Accelerated Climate Modeling for Energy: Project Strategy and Initial Implementation Plan. ACME Council, Department of Energy, 25 p. http://climatemodeling.science.energy.gov/sites/default/files/publications/acme-project-strategy-plan.pdf

[2] Overpeck, J.T., Meehl, G.A., Bony, S. and Easterling, D.R. (2011) Climate Data Challenges in the 21st Century. *Science*, **331**, 700-702. http://dx.doi.org/10.1126/science.1197869

[3] Meehl, G.A., Moss, R., Taylor, K.E., Eyring, V., Stouffer, R.J., Bony, S. and Stevens, B. (2014) Climate Model Intercomparison: Preparing for the Next Phase. *Eos, Transactions American Geophysical Union*, **95**, 77-78. http://dx.doi.org/10.1002/2014EO090001

[4] Taylor, K.E., Stouffer, R.J. and Meehl, G.A. (2012) An Overview of CMIP5 and the Experiment Design. *Bulletin of the American Meteorological Society*, **93**, 485-498. http://dx.doi.org/10.1175/BAMS-D-11-00094.1

[5] Williams, D.N., Lautenschlager, M., Balaji, V., Cinquini, L., DeLuca, C., Denvil, S., Duffy, D., Evans, B., Ferraro, R., Juckes, M. and Trenham, C. (2015) Strategic Roadmap for the Earth System Grid Federation. 2015 *IEEE International Conference on Big Data*, Santa Clara, 29 October-1 November 2015, 2182-2190. http://dx.doi.org/10.1109/BigData.2015.7364005

[6] Williams, D.N., Balaji, V., Cinquini, L., Denvil, S., Duffy, D., Evans, B., Ferraro, R., Hansen, R., Lautenschlager, M. and Trenham, C. (2016) A Global Repository for Planet-Sized Experiments and Observations. *Bulletin of the American Meteorological Society*, **97**.

[7] Williams, D.N. (2016) Better Tools to Build Better Climate Models. *Eos*, **97**.

[8] Santos, E., Poco, J., Wei, Y. X., Liu, S.S., Cook, B., Williams, D.N., *et al.* (2013) UV-CDAT: Analyzing Climate Datasets from a User's Perspective. *Computing in Science & Engineering*, **15**, 94-103. http://dx.doi.org/10.1109/MCSE.2013.15

[9] Gropp, W., Lusk, E. and Skjellum, A. (1996) A High-Performance, Portable Implementation of the MPI Message Passing Interface. *Parallel Computing*, **22**, 789-828. http://dx.doi.org/10.1016/0167-8191(96)00024-5

[10] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W. and Dongarra, J.J. (1998) MPI—The Complete Reference: Volume 1, The MPI Core. MIT Press, Cambridge.

[11] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W. and Snir, M. (1998) MPI—The Complete Reference: Volume 2, The MPI-2 Extensions. MIT Press, Cambridge.

[12] Layton, J.B. (2009) Caos NSA and Perceus: All-in-One Cluster Software Stack. *Linux Magazine*, 5 February 2009.

[13] Georgiou, Y. (2010) Contributions for Resource and Job Management in High Performance Computing. Thesis, Universite Joseph Fourier, France.

[14] Dalcin, L., Paz, R. and Storti, M. (2005) MPI for Python. *Journal of Parallel and Distributed Computing*, **65**, 1108-1115. http://dx.doi.org/10.1016/j.jpdc.2005.03.010

[15] Dalcin, L., Paz, R., Storti, M. and D'Elia, J. (2008) MPI for Python: Performance Improvements and MPI-2 Extensions,

Journal of Parallel and Distributed Computing, **68**, 655-662. http://dx.doi.org/10.1016/j.jpdc.2007.09.005

[16] Dalcin, L., Kler, P., Paz, R. and Cosimo, A. (2011) Parallel Distributed Computing Using Python. *Advances in Water Resources*, **34**, 1124-1139. http://dx.doi.org/10.1016/j.advwatres.2011.04.013

[17] Yoo, A., Jette, M. and Grondona, M. (2003) SLURM: Simple Linux Utility for Resource Management, Job Scheduling Strategies for Parallel Processing. *Lecture Notes in Computer Science*, **2862**, 44-60. http://dx.doi.org/10.1007/10968987_3

[18] Jette, M. and Grondona, M. (2003) SLURM: Simple Linux Utility for Resource Management. *Proceedings of Cluster World Conference and Expo*, San Jose, June 2003.

[19] Balle, S.M. and Palermo, D. (2007) Enhancing an Open Source Resource Manager with Multi-Core/Multi-Threaded Support. *Lecture Notes in Computer Science*, **4942**, 37-50. http://dx.doi.org/10.1007/978-3-540-78699-3_3

[20] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S. and Stoica, I. (2010) Spark: Cluster Computing with Working Sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (*HotCloud*'10), USENIX Association, Berkeley, 10 p.