Evaluating MPI Performance and Scalability in Supercomputing Environments

Morgan Olivia Newton
High-Performance Computing, Code 606.2

NASA Center for Climate Simulation
NASA Goddard Space Flight Center
Greenbelt, MD
Summer 2024

NASA Center for Climate Simulation
Goddard Space Flight Center

Contents

1. Introduction

1.1. GMAO and GEOS Systems

The Global Modeling and Assimilation Office (GMAO) conducts modeling and assimilation activities to support NASA's Earth Observation missions, utilizing data from past missions while aiding in planning for our future. The Goddard Earth Observing System (GEOS) model is key to these activities, consisting of multiple components designed for various Earth Science applications. The GEOS Earth System Model spans a range of dynamical, physical, chemical, and biological processes [1]. This study aims to investigate MPI performance anomalies in high-performance computing (HPC) for climate and weather forecasting, with the goal of optimizing these systems for future use.

1.2. MPI Performance and Supercomputing Scalability

MPI (Message Passing Interface) is a standard way for different parts of a program to communicate with each other in a supercomputing environment. When running GEOS models, a significant drop in MPI performance is observed once the number of tasks per node exceeds 64 on AMD Milan architecture, with significant increases in both latency and elapsed time.

OSU (Ohio State University) Micro-benchmarks were conducted across three different systems to compare MPI performance, specifically focusing on these metrics. Scalability within supercomputing is crucial because it allows systems to handle larger tasks and additional data efficiently. This ensures that complex simulations and models run faster and more accurately, ultimately improving scientific research and analysis. By evaluating MPI performance, areas for improvement in communication efficiency can be identified, which in turn enhances scalability, optimizes resource utilization, and improves overall system performance.

2. Systems and Architecture

2.1. The NASA Center for Climate Simulation

The NASA Center for Climate Simulation (NCCS) provides global HPC resources to scientists and researchers. These resources include the Discover supercomputer, an important component of NCCS. Discover features several hardware architectures, most notably, AMD Milan nodes which deliver nearly 8.1 petaflops of computational power [2]. This evaluation seeks to characterize the performance and scalability of these nodes, specifically evaluating their effectiveness with GEOS climate code, while also exploring potential performance anomalies with MPI calls.

2.2. Microsoft Azure and AWS

The Science Managed Cloud Environment (SMCE) integrates Microsoft Azure and Amazon Web Services (AWS) to offer scalable cloud computing to users [3]. Both HPC clusters and cloud computing provide access to powerful hardware on demand, but they differ in how users access and manage these resources.

2.3. NASA Advanced Supercomputing

Sponsored by the High-End Computing Capability project, the NASA Advanced Supercomputing (NAS) facility at NASA's Ames Research Center features three supercomputers: Pleiades, Electra, and Aitken. Aitken, NASA's most powerful supercomputer (10.76 petaflops), is housed in the Modular Supercomputing Facility at Ames, featuring Intel Xeon Gold, AMD Rome, and AMD Milan nodes [4].

2.4. System Evaluation Overview

A comparative analysis was performed to provide insights into the optimal use of these diverse computing environments for supercomputing tasks, particularly with GEOS climate code. OSU micro-benchmarks were conducted on Discover, Azure, AWS, and Aitken using AMD Milan architecture to compare the performance and scalability of all three systems. The comparison included examining the operating systems (OS) used in these environments: NCCS cloud systems Azure and AWS run on Ubuntu, NAS Aitken operates on Red Hat Enterprise Linux (RHEL), and NCCS Discover uses SUSE

Linux Enterprise Server (SLES). This analysis aims to identify any performance variances due to the underlying operating systems.

3. Benchmarking

3.1. OSU Micro-benchmarks

The OSU Micro-benchmarks are a suite of tests used to evaluate the performance of MPI. These benchmarks measure latency, bandwidth, multiple-bandwidth and message rates. They serve as essential tools for identifying performance bottlenecks, comparing different systems, interconnects, and MPI implementations [5]. In this study, the following benchmarks were used to analyze performance: osu_allgather, osu_allgatherv, osu_allreduce, osu_barrier, osu_bcast, osu_gather, osu_gatherv, osu_reduce, osu_reduce_scatter, osu_scatter, and osu_scatterv.

To support GEOS efforts to improve scalability, MPI allgather and allgatherv operations were the main focus. These operations face three scaling limits: a small word injection limit (the rate of small message injection), an injection bandwidth limit (the maximum data transfer rate), and a network bisection limit (the maximum throughput the network can handle when evenly split). These limits cause congestion and decreased performance as messages and core counts increase. MPI allgather and allgatherv operations take a piece of data from every process and share the entire collection back to every process, making them a vital part of efficient parallel computing.

3.2. MPI Tuning

MPI tuning consists of several configurations designed to optimize communication performance. Tuning optimizes system performance by tailoring configurations to specific workloads.

I. No Tuning
- Uses default MPI settings without any added adjustments.

II. OFI Tuning
- Involves setting the environment variable I_MPI_FABRICS=ofi.
- Utilizes OpenFabrics Interfaces for efficient inter-node communication.

III. GMAO + OFI Tuning
- Uses I_MPI_FABRICS=ofi and incorporates additional environment variables to enhance performance developed by the GMAO.

IV. GMAO + SHM + OFI Tuning
- Uses I_MPI_FABRICS=shm:ofi to combine shared memory (shm) and OFI for communication.

3.3. Parameters

All systems used constant parameters with minor deviations based on version and architecture availability. Tuning and operating system evaluations used 8192 message size for all systems. Message size evaluations used 2048, 4096, 8192 message sizes.

System	Architecture	OS	Compiler	MPI	Message Size	nodes : tasks per node
Discover	AMD Milan	SLES	GCC 12.3	MPI 2021.13	8192	50:2, 50:4, 50:8, 50:16, 50:32, 50:46, 50:64, 50:126, 64:126
AWS	AMD Milan	Ubuntu	GCC 12.3	MPI 2021.13	8192	50:2, 50:4, 50:8, 50:16, 50:32, 50:46, 50:64, 50:96, 64:96
Aitken	AMD Milan	Red Hat	GCC 12.3	MPI 2021.13	8192	50:2, 50:4, 50:8, 50:16, 50:32, 50:46, 50:64, 50:126, 64:126

4. Setup and Build

4.1. Compilation Process

The process of compiling across each system involved several key steps with minor variations for each environment. Initially, access to each system was established using SSH with the appropriate keys and user credentials. Once connected, necessary directories and files were transferred from the local machine

to the remote systems. On the remote systems, the transferred files were extracted to prepare for the build process. Next, the required modules for GCC and MPI were loaded, typically involving sourcing available modules and Spack environments. Resource allocation for the build process was performed using system-specific commands to allocate nodes, tasks and other options. The benchmark source files were then extracted, and the build environment was configured using the appropriate compilers. The benchmarks were then compiled, leveraging multiple parallel jobs to expedite the process. Finally, any temporary or intermediate files created during the build were cleaned up to maintain a neat working environment. Each build was thoroughly documented in Confluence to streamline future work.

4.2. Comparative System Analysis

All systems utilized unique hardware and software configurations, contributing to slight variations in MPI performance and influencing overall performance metrics. Discover, Azure and AWS employ Slurm, an open-source job scheduler known for its scalability and flexibility in managing workloads, while NAS uses PBS (Portable Batch System), recognized for its stability and longevity. Additionally, Discover, Azure and AWS differ in their approach to module loading. Discover and NAS use traditional software modules, providing a user-friendly experience. Azure and AWS use both modules and the Spack package manager, which required a more in-depth approach for software installation.

4.3. Challenges

On NAS, PBS presented significant challenges. The entire sbatch command required conversion into a qsub command. Several unique flags also needed to be added to correctly convert to the new system. PBS provides an environment variable for tasks per node (*$NCPUS*), but lacks environment variables for nodes and total tasks. To account for this inconsistency, custom variables were built using the *$PBS_NODEFILE* variable, which lists all nodes allocated by the PBS scheduler. The variable *$NO_NODES* calculates the number of unique nodes by sorting *$PBS_NODEFILE* and counting every unique hostname. The variable *$TOTAL_TASKS* determines the total number of tasks by counting the total lines in *$PBS_NODEFILE*, representing all allocated cores. These manual adjustments enabled build scripts to function accurately on the new system.

Azure did not have the necessary compiler and MPI versions, requiring the creation of a customized Spack environment to accommodate for system variations. When attempting to use the required Intel

MPI version, system-wide issues were encountered that necessitated escalation to NCCS, Azure and Intel experts. The error occurred when attempting to use mpirun (a command for launching MPI applications) in the build script. The job entered the run state but never executed, and no error information was produced. This unusual terminal hang made the issue extremely difficult to identify and diagnose. The issue was escalated to Intel experts and is currently being addressed. Although the discovery of the software bug is beneficial, it has unfortunately paused progress with Azure.

Navigating AWS presented its own set of challenges. While GCC and Intel MPI modules loaded without issue, several problems were encountered regarding proper resource allocation due to account limits. Both Azure and AWS accounts were initially restricted, requiring a manual increase in usage limits. Additionally, the tasks per node had to be adjusted to a maximum of 96, as the system architecture inherently did not support the originally intended 126 tasks per node, necessitating a reconfiguration of the benchmark parameters to align with these structural limitations.

Additionally, the structure of the AWS cloud system as a whole posed unique difficulty. The first few benchmarks resulted in segmentation faults, which occur when a program attempts to access an invalid memory location, often due to resource allocation issues. The issue was narrowed down to an application problem related to the '-x' flag associated with the benchmark. The '-x' flag is used to denote which iteration to start gathering data from, typically set to 1 to ensure that no warm-up iterations are skipped. The root cause of the problem remains unclear, but removing the '-x' flag from the script resolves the issue. This adjustment allowed all data to be collected without errors, though it will omit startup costs in the average latency assessment, potentially affecting the accuracy of the performance measurements.

5. Results

5.1. Scalability Analysis

Benchmark performance metrics track OSU benchmarks, compiler settings, MPI configurations, tuning parameters, node counts, tasks per node, total tasks, message size, and iterations. In addition, key metrics include average, minimum, and maximum latency, as well as elapsed time. The analysis primarily focused on average latency and elapsed time to investigate overall performance.

5.2. Message Sizing

As the number of tasks per node increases, a linear progression in latency is observed. Latency increases significantly past 64 tasks per node, indicating that delays grow steadily with workload size. Comparing message sizing with osu_allgather and osu_allgatherv benchmarks on Discover Milan architecture reveals that all three message sizes (2048, 4096, and 8192) confirm an increase in average latency after 64 tasks per node, indicating poor scalability.

As seen in Figures 1 and 2, average latencies in the osu_allgather benchmark reach up to 52,210.94 microseconds (µs) for 126 tasks per node at a 2048 message size, while in the osu_allgatherv benchmark, they reach 42,353.53 µs for the same configuration. Although the 8192-message size shows a more substantial increase, all message sizes tested exhibit an upward trend in average latency. All message sizes tested show potential leveling off in average latency at around 60000 µs, suggesting that the impact on latency may stabilize at higher ranks.

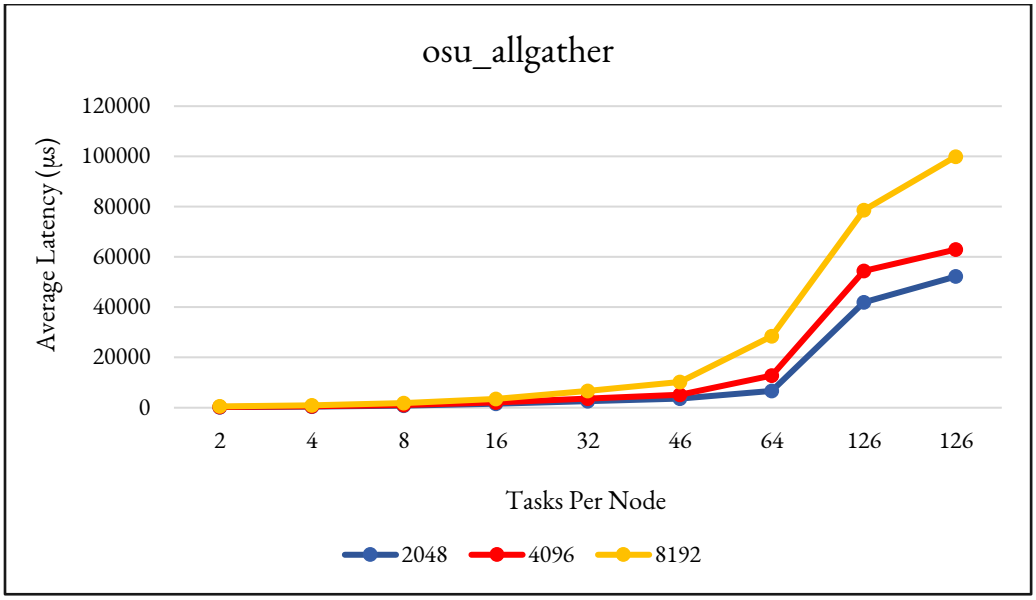

Figure 1: OSU Allgather Benchmark: Average latency increases significantly across all message sizes after 64 tasks per node on Discover Milan architecture for message sizes 2048, 4096, and 8192.

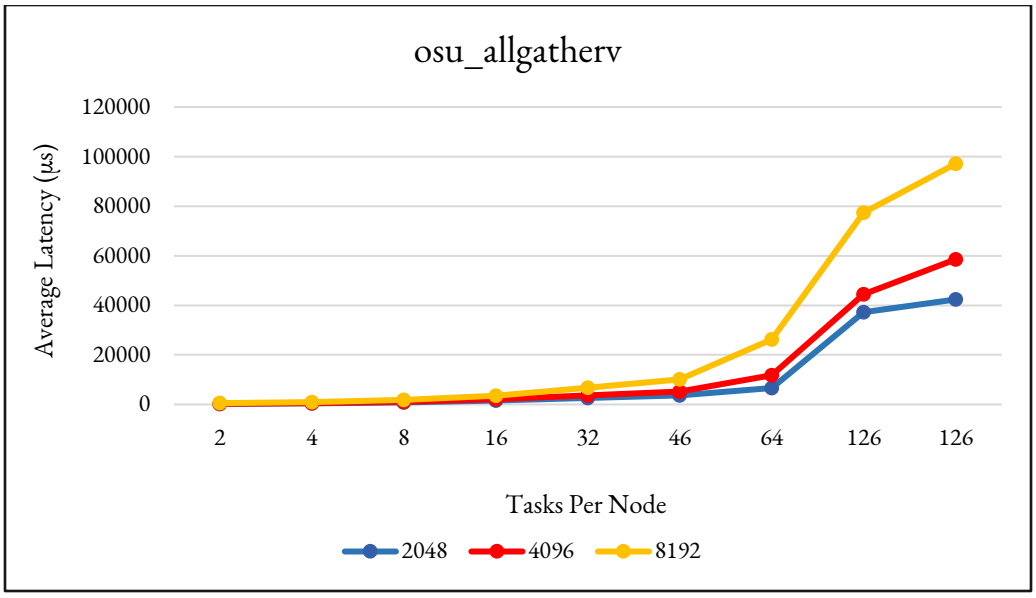

Figure 2: OSU Allgatherv Benchmark: Message sizes 2048, 4096, and 8192 show potential leveling off in average latency at around 60,000 µs, suggesting that the impact on latency may stabilize at higher ranks.

5.3. Tuning

As seen in Figures 3, 4, 5, and 6, average latency increases significantly as ranks increase across all systems, but this increase is less severe when certain tuning parameters are applied to Discover. Although applying GMAO+OFI and GMAO+SHM+OFI tuning improves scalability on Discover, particularly for allgather and allgatherv operations, Aitken experiences a minor regression with these tuning parameters.

The reductions in average latency observed are influenced by both the tuning's positive impact on Discover and Aitken's performance degradation. To fully understand whether the reduction in the performance gap between Discover and Aitken is due to Discover's improvement or Aitken's regression, it is necessary to analyze the absolute latencies of both systems under different tuning parameters. Aitken's worsening performance plays a significant role in narrowing the gap, as average latency increases notably under these tuning settings.

Although Aitken initially demonstrates better performance without tuning, its performance becomes less consistent when tuning parameters are applied, leading to significant regression in many cases. This inconsistency suggests that while Aitken's performance is more stable without tuning, it struggles to

maintain that stability under tuned conditions. Additionally, OFI appears to have the least impact on latency across all systems, especially for AWS, where scaling issues persist regardless of tuning. Tuning analysis used 8192 message sizing across all systems. For larger configurations, AWS had an architecture restriction of 96 max ranks.

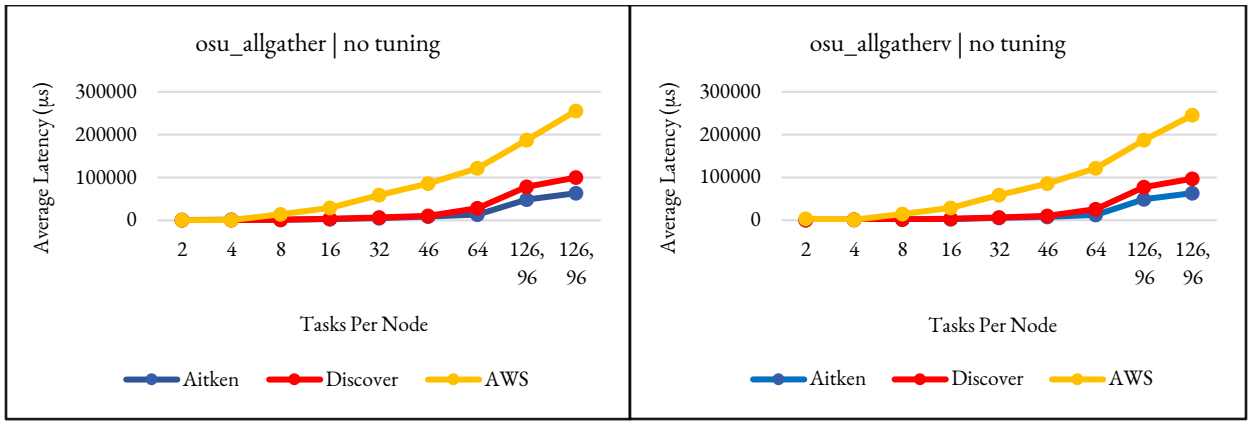

Figure 3: OSU Allgather and Allgatherv Benchmarks - no tuning across Aitken, Discover and AWS.

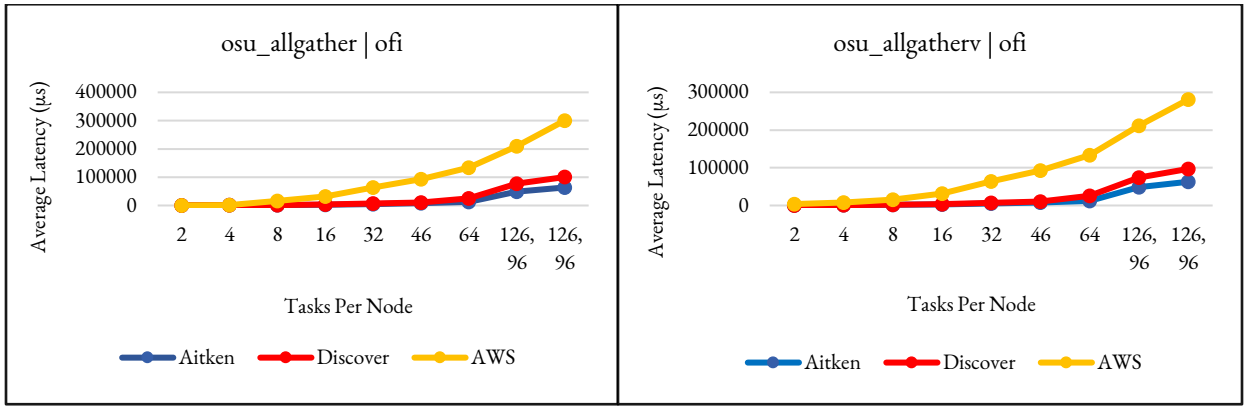

Figure 4: OSU Allgather and Allgatherv Benchmarks - OFI tuning across Aitken, Discover and AWS.

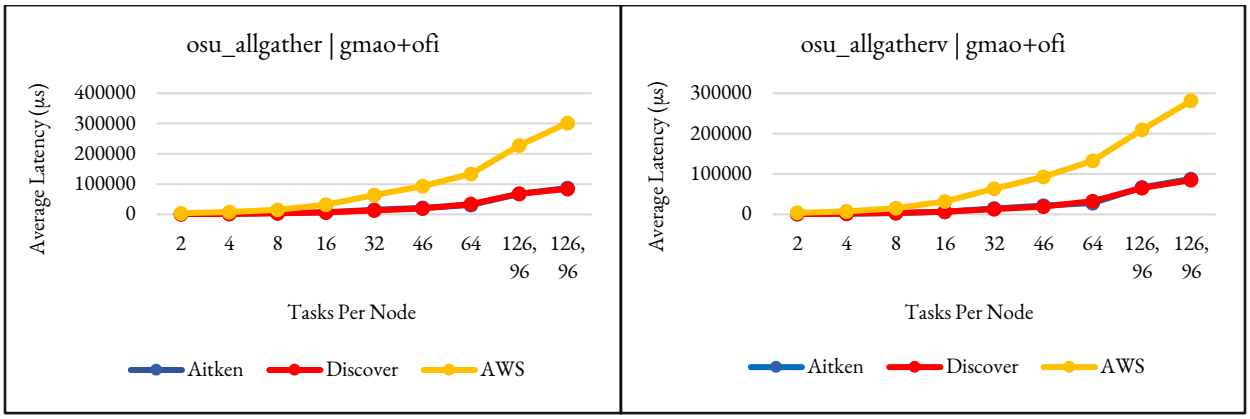

Figure 5: OSU Allgather and Allgatherv Benchmarks – GMAO+OFI tuning across Aitken, Discover and AWS.

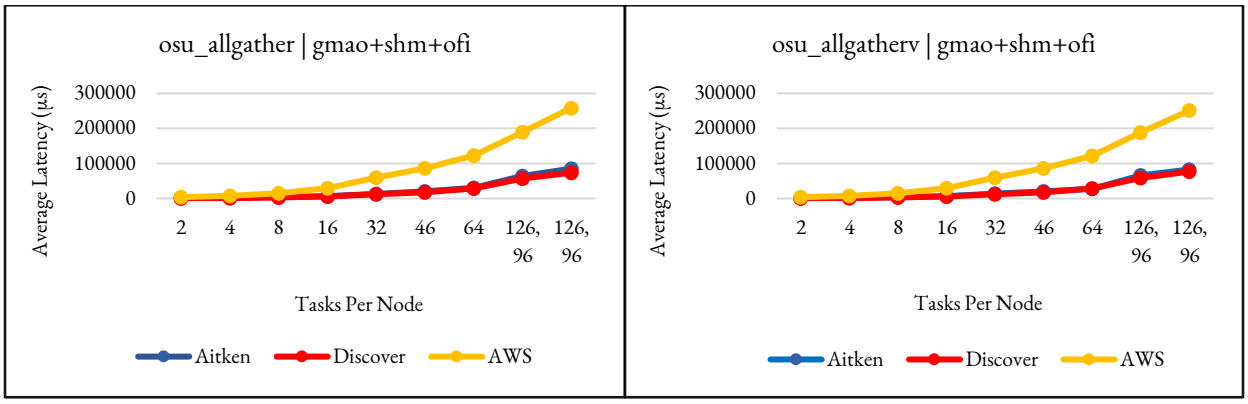

Figure 6: OSU Allgather and Allgatherv Benchmarks – GMAO+SHM+OFI tuning across Aitken, Discover and AWS.

6. Conclusions

6.1. Operating Systems

The evaluation of Discover (SLES), AWS (Ubuntu), and Aitken (Red Hat) operating systems reveal no significant performance differences between systems regarding scalability. Benchmarking shows that each system exhibits a consistent increase in average latency once the number of tasks per node exceeds 64. This pattern confirms that the observed latency issues are not attributable to operating system anomalies. Consequently, these findings rule out operating systems as a factor, allowing for a focus on other potential causes.

6.1. Aitken Summary

When comparing Discover, AWS, and Aitken directly, NAS Aitken demonstrates better scalability without tuning, exhibiting lower average latency in both the osu_allgather and osu_allgatherv benchmarks. As seen in Figures 3 and 4, Aitken performs better than Discover by almost a factor of 2 as the number of tasks per node increases, particularly at higher configurations with 126 tasks per node. Without tuning, all benchmarks show slightly better performance on Aitken.

It is important to note that NAS has Simultaneous Multi-Threading (SMT) [6], also known as Hyper-Threading, enabled on the Aitken cluster. This technology allows each physical core to handle two threads, effectively doubling the number of cores visible to the operating system [7]. In a two-socket configuration, this means the Linux OS recognizes 256 cores instead of 128. While this can increase parallel efficiency, its benefits are subjective; degradation may occur in CPU-intensive tasks. The NCCS traditionally disables this technology, as it is less effective for workloads that fully utilize CPU resources, such as physics computations like GEOS. Therefore, this technology could theoretically contribute to the observed performance differences between systems.

6.2. AWS Summary

AWS, in contrast, performs noticeably worse compared to both Discover and Aitken. AWS shows a dramatic and concerning increase in average latency as tasks per node increases. This is interesting given that startup iterations were omitted and the Milan architecture on AWS restricts tasks per node to only

96 based on the chip architecture on AWS. The expectation was that AWS would perform better due to these differences, yet results indicate otherwise. These findings suggest that factors other than operating system or architecture limitations are contributing to the observed performance discrepancies on AWS. All things considered, all systems show a clear issue with scalability.

Results across all OSU micro-benchmarks show that AWS consistently underperforms above 4000-5000 total tasks, compared to Discover and Aitken. Notably, InfiniBand and Ethernet are different networking technologies for data transfer. InfiniBand provides high bandwidth and low latency, while Ethernet is robust but generally higher latency. AWS uses Ethernet, which may result in higher latency and lower bandwidth compared to Discover and Aitken's InfiniBand. This suggests that AWS's use of Ethernet networking substantially impacts overall performance, leading to subpar results at scale.

6.3. Implications

The implications of this work are crucial for supercomputing and directly benefit the GMAO. By confirming that operating system differences do not significantly impact performance, the research eliminates one potential variable in optimizing supercomputing environments. These methods can be applied to future studies and evaluations, fostering more efficient and effective research. Additionally, the research also helps refocus software developers' efforts on optimizing their MPI approaches now that operating system differences have been ruled out as a contributing factor.

6.4. Future Work

Future work should focus on implementing smarter MPI allgather methods, such as reducing the number of messages sent per rank or utilizing fan-in/fan-out techniques to gather and distribute data more efficiently. These techniques involve collecting data in stages (fan-in) and then distributing it in stages (fan-out). This method could optimize communication patterns to reduce network congestion and improve overall performance.

Lastly, resolving the outstanding issues with Azure would facilitate a direct performance comparison between cloud systems and would assist engineers in understanding how these systems differ compared to Discover and Aitken. Moving forward, all future scalable units should undergo the same rigorous testing processes to verify their performance and scalability.

NASA Center for Climate Simulation
Goddard Space Flight Center

Acknowledgments

Laura Carriere – Thank you for your support, trust, and patience throughout the summer. Most notably, your willingness to grant me this life-changing opportunity after over a year of persistent emails!

Bruce Pfaff – Thank you for your invaluable insights, for graciously welcoming me into the NCCS team with open arms and most importantly, for introducing me to authentic Maryland crab picking.

William Woodford – Thank you for all of the countless hours you contributed to troubleshooting various systems, teaching me dozens of terminal commands and shortcuts, and for being an absolute pleasure to work with in every aspect.

Hoot Thompson – Thank you for your help with setting up cloud computing accounts for Azure and AWS, managing Spack modules, troubleshooting persistent system issues, and painstakingly fine-tuning system resource limits.

Nicko Acks – Thank you for your assistance in overcoming technical difficulties with my laptop and cluster account during my first week with NCCS. I would not have been able to accomplish any of this work without your guidance.

Lastly, Matt Thompson – thank you for going out of your way to help me troubleshoot Intel MPI on Azure and for sharing numerous valuable technical resources throughout the summer that have greatly benefited my work.

References

[1] Global Modeling and Assimilation Office (GMAO). GEOS Systems.
url: https://gmao.gsfc.nasa.gov/GEOS_systems

[2] NASA Center for Climate Simulation (NCCS). (n.d.). NASA Center for Climate Simulation (NCCS). url: https://www.nccs.nasa.gov/systems

[3] NASA Scientific and Mission Computing Environment (SMCE). (n.d.). NASA Scientific and Mission Computing Environment (SMCE). url: https://smce.nasa.gov/overview

[4] NASA Advanced Supercomputing (NAS) Division. (n.d.). NASA Advanced Supercomputing (NAS) Division.
url: https://www.nas.nasa.gov/hecc/services/supercomputing_systems_service.html and https://www.nas.nasa.gov/hecc/resources/aitken.html

[5] Ohio State University. (2001-2016). OSU Micro-Benchmarks (OMB) [Software]. Network-Based Computing Laboratory, The Ohio State University. Available under BSD License. Developed by the Network-Based Computing Laboratory (NBCL), headed by Professor Dhabaleswar K. (DK) Panda.
url: https://www.github.com/forresti/osu-micro-benchmarks?tab=License-1-ov-file

[6] AMD Corporation. (n.d.). Glossary of Terms. url:
https://www.amd.com/content/dam/amd/en/documents/resources/glossary-of-terms.pdf

[7] Intel Corporation. (n.d.). Intel® Hyper-Threading Technology.
url: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html