Enabling Low-Overhead HT-HPC Workflows at Extreme Scale using GNU Parallel

Ketan Maheshwari*, William Arndt†, Ahmad Maroof Karimi*, Junqi Yin*
Frédéric Suter*, Seth Johnson*, Rafael Ferreira da Silva*
*Oak Ridge National Laboratory, Oak Ridge, TN, USA
†Lawrence Berkeley National Laboratory, Berkeley, CA, USA
{maheshwarikc,karimiahmad,yinj,suterf,johnsonsr,silvarf}@ornl.gov, warndt@lbl.gov

***Abstract*—GNU Parallel is a versatile and powerful tool for process parallelization widely used in scientific computing. This paper demonstrates its effective application in high-performance computing (HPC) environments, particularly focusing on its scalability and efficiency in executing large-scale high-throughput high-performance computing (HT-HPC) workflows. Through real-world examples, we highlight GNU Parallel's performance across various HPC workloads, including GPU computing, container-based workloads, and node-local NVMe storage. Our results on two leading supercomputers, OLCF's Frontier and NERSC's Perlmutter, showcase GNU Parallel's rapid process dispatching ability and its capacity to maintain low overhead even at extreme scales. We explore GNU Parallel's application in massive parallel file transfers using a scheduled Data Transfer Node (DTN) cluster, emphasizing its broad utility in diverse scientific workflows. Beyond its direct application as a viable workflow manager, GNU Parallel can be employed in conjunction with other workflow systems as a "last-mile" parallelizing driver and as a quick prototyping tool to design and extract parallel profiles from application executions. We then argue that the potential for GNU Parallel to transform workflow management at extreme scales is substantial, paving the way for more efficient and effective scientific discoveries.**

***Index Terms*—Scientific Workflows, Distributed Computing, GNU Parallel, High-Throughput Computing**

I. Introduction

In the modern era of scientific research, the capacity to manage and execute large-scale computations and data analyses has become a cornerstone of scientific discovery, driving innovations in fields ranging from genomics, to high-energy physics, to climate modeling [1], [2]. High-throughput scientific workflows, in particular, have become increasingly vital as they accelerate the pace of scientific discovery by enabling researchers to process vast amounts of data efficiently and effectively at extreme scale [3], [4]. One compelling example is the experiments conducted at the Large Hadron Collider (LHC) at CERN, which generate massive amounts of data through numerical simulations and comparisons with data produced by detectors. These experiments, such as the discovery of the Higgs boson and the observation of gravitational waves, highlight the necessity for high-throughput computing to handle millions of jobs submitted monthly, underscoring the critical need for efficient workflow management [5]. However, as the scale of scientific computations grows, so does the complexity of managing these workflows.

Despite the plethora of workflow management systems available today [6], a recent benchmark study has highlighted significant challenges associated with orchestrating large-scale workflows on high-performance computing (HPC) systems—the overhead involved in managing such workflows can be substantial, often impeding performance [7]. This overhead includes the time and resources required for scheduling, data transfer, synchronization, and error handling, which can become significant as the number of tasks and the scale of computation increase. As we transition towards exascale computing, it is imperative to explore alternative solutions that can execute high-throughput high-performance computing (HT-HPC [8]) workflows with minimal overhead. This transition presents both an opportunity and a challenge: to harness the power of these advanced systems effectively, we must minimize the orchestration overhead to ensure that the computational resources are utilized efficiently.

GNU Parallel [9] is a versatile command-line tool designed to execute jobs in parallel across one or more compute nodes. It excels at running multiple instances of programs or scripts concurrently, effectively distributing the workload across all available resources. This capability allows users to parallelize tasks such as data processing, simulations, and analyses, resulting in significant performance improvements with minimal overhead. The tool's flexibility and ease of use make it especially suitable for HT-HPC workflows, where both efficiency and simplicity empowers users with high productivity.

In this paper, we demonstrate the compelling advantages of adopting GNU Parallel for HT-HPC workflows. We highlight its simplicity and efficiency and show its transformative potential for executing scientific workflows at an extreme scale. Through detailed analysis and real-world use cases, we illustrate how GNU Parallel can integrate seamlessly into various computational environments, providing researchers with a powerful tool to achieve their computational objectives

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

while maintaining a straightforward and efficient workflow management process.

In this work, we make the following contributions:

- We demonstrate the effective application of GNU Parallel to optimize the utilization of CPUs, GPUs, NVMe storage, and large-scale storage systems on leading HPC platforms like OLCF's Frontier and NERSC's Perlmutter. By efficiently managing these resources, GNU Parallel enables HT-HPC workflows to run smoothly and effectively at extreme scale.
- We provide a detailed analysis of overhead metrics, showcasing GNU Parallel's scalability and efficiency across thousands of compute nodes, with empirical evidence highlighting its ability to maintain low overhead even as the scale of computation increases.
- We present practical vignettes and real-world applications to illustrate the versatility and applicability of GNU Parallel in diverse HT-HPC workloads at scale.
- We present novel solutions for common computational and I/O challenges, validated through large-scale workflow applications on Frontier and Perlmutter supercomputers. These solutions include techniques for data prefetching, concurrent GPU utilization, and massive data transfer, showcasing the innovative potential of GNU Parallel in addressing complex workflow requirements.

This paper is structured as follows: In Section II, we review the background and related work, discussing the foundational concepts and existing tools relevant to HT-HPC workflows. In Section III, we present the performance benchmarks of GNU Parallel, detailing experiments conducted on OLCF's Frontier and NERSC's Perlmutter supercomputers. In Section IV, we showcase various applications of GNU Parallel, including a motivating use-case for data fetch-process workflows, massive Darshan log processing, the FORGE large language model, GPU computing with Celeritas, and data motion strategies. Finally, Section V concludes the paper with a summary of our findings and their implications for future research in efficient workflow management.

II. Background and Related Work

Workflows are the cornerstone of modern scientific research, enabling the integration and automation of complex computational processes essential for advancing discoveries. They have been pivotal in securing recent Gordon Bell prizes [10], [11] and serve as a foundational technology for accelerating AI and ML applications on HPC systems [4], [8]. Consequently, numerous workflow management systems have been developed to support these advancements [6]. However, as workflows evolve and their capabilities expand, the architecture of their components becomes increasingly intricate. A recent benchmarking study [7] highlighted that the overhead associated with orchestrating workflow activities at scale can significantly hinder performance. For instance, Fig 10 in [7] shows, for different workflows scheduled (on the Summit supercomputer) with Swift/T but with no data transfers and no computation (i.e., just launching the tasks) the overhead is 500 seconds for 50,000 tasks and up to 5,000 seconds for 100,000 tasks of the BLAST workflow. This observation underscores the need for a low-overhead, straightforward tools that can efficiently manage workflows at an extreme scale. Therefore, this work aims to revisit fundamental principles to identify and leverage simplistic yet powerful tools that enhance workflow management efficiency.

GNU Parallel [9] is a powerful Linux command-line tool designed to launch and manage multiple processes in parallel, working seamlessly with traditional Linux utilities and constructs such as pipes, environment variables and standard streams. It provides close to 100 command-line options, allowing extensive customization of parallel processing workflows. By efficiently distributing workloads across available computational resources, it significantly reduces the complexity and overhead of workflow orchestration. GNU Parallel is not only effective for exploiting ad-hoc and opportunistic parallelism in scientific applications but also serves as an adept prototyping tool for rapidly extracting and optimizing an application's parallel profile. Integrating GNU Parallel into HT-HPC environments has the potential to enable the efficient management of complex scientific workflows, resulting in faster execution times and higher productivity.

Several studies have leveraged GNU Parallel for various HPC tasks, but these works primarily focus on optimizing specific applications rather than workflow management. For instance, Mueller-Bady et al. [12] introduced the Multijob framework, which uses GNU Parallel for the efficient distribution of evolutionary algorithms in parameter tuning. This framework simplifies the configuration and execution of experiments on heterogeneous computing clusters, highlighting the utility of GNU Parallel in handling job distribution, pausing, resuming, and estimating completion times. Similarly, Klenk and Spiteri [13] utilized GNU Parallel to manage large-scale hydrological simulations, comparing its performance against other job submission tools like Slurm. They found that while GNU Parallel effectively balanced computational loads across processors, it did not specifically address workflow orchestration complexities. Additionally, the POAP pipeline developed by Samdani and Vetrivel [14] exemplifies the use of GNU Parallel in a multithreaded virtual screening process, optimizing ligand preparation and docking tasks. Although POAP demonstrates significant scalability and efficient resource utilization, it similarly does not address the broader challenges of managing end-to-end scientific workflows, such as data management and minimizing overhead at extreme scales. These studies demonstrate the effectiveness of GNU Parallel in parallel job execution but do not explore its potential for managing the comprehensive requirements of scientific workflows at scale.

Many research computing centers offer GNU Parallel as a managed software module and actively encourage users to leverage it for parallel job execution. However, to the best of our knowledge, no extensive studies have been conducted to evaluate its performance comprehensively over HPC systems.

In this work, we address this gap by conducting an in-depth study of GNU Parallel's capabilities, providing a detailed performance analysis on a large-scale installation. This study aims to highlight the benefits and potential of GNU Parallel, encouraging its broader adoption for managing a wide variety of scientific workflows efficiently.

III. Performance Benchmarks

To comprehensively evaluate the performance of GNU Parallel, we conducted a series of experiments focusing on scalability, GPU efficiency, stress testing, and containers launching performance. These experiments were performed on two cutting-edge supercomputers: OLCF's Frontier, a leadership class exascale HPC system, and NERSC's Perlmutter, an HPC system designed to support both CPU and GPU workloads efficiently. Hereafter, we detail the various tests and their outcomes.

Scalability Runs on Frontier. We conducted weak scaling runs on Frontier by launching one instance of GNU Parallel per node on up to 9,000 nodes (96% of Frontier). The task to execute involved running a simple one-liner bash script that records the hostname and timestamp to standard output for validation and performance measurements. Each node executed 128 parallel instances of this script, corresponding to its 128 CPU cores (64 dual-threaded). We distributed the tasks across nodes via a driver script that cyclically distributes input parameters to the available nodes in a Slurm job allocation. As illustrated in the code shown in listing 1 this driver script utilizes two Slurm environment variables to determine the total number of nodes and the unique node that the current job is running.

```
1 #!/bin/bash
2 cat $1 | \
3 awk -v NNODE="$SLURM_NNODES" \
4     -v NODEID="$SLURM_NODEID" \
5     'NR % NNODE == NODEID' | \
6 parallel -j128 ./payload.sh {}
```

Listing 1. A driver script that distributes tasks evenly to available nodes in a Slurm job (invoked within a Slurm job as `./driver.sh inputs.txt`)

The script is invoked with the inputs as first command line argument that is passed to an `awk` one-liner as shown in line 2 and 3. The Slurm environment variables are passed to awk variables in lines 3 and 4. The `awk` expression in line 5 will evaluate to true for one unique node for each input line resulting in an even distribution of inputs across the nodes. Finally, line 6 will invoke the application with GNU Parallel passing the arguments thus collected in the {} construct.

The standard output was initially written to the node-local NVMe for I/O efficiency and to avoid writing small files to the Lustre filesystem, adhering to best practices. At the end of each run, the aggregated output was transferred to the persistent Lustre filesystem. The start and end times of all runs were collected, and the difference between the earliest start and latest end times is shown in Fig. 1.

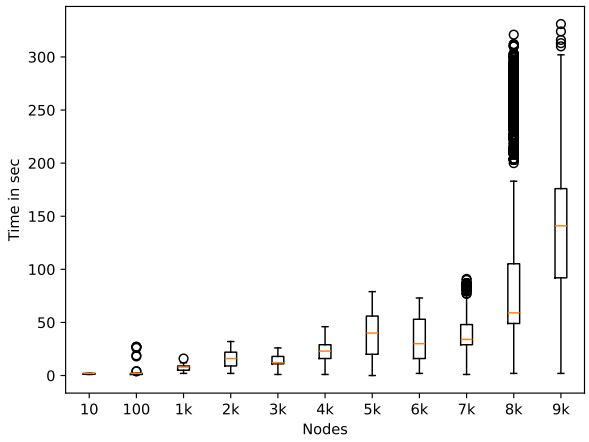

Fig. 1. Weak scaling on Frontier. Each node executed 128 parallel instances of a simple bash script, with standard output initially written to node-local NVMe before being transferred to the Lustre filesystem. Half of the processes completed in less than a minute, and 75% completed in less than two minutes with 8,000 nodes. Greater variance was observed in 9,000-node runs due to outlier nodes, possibly caused by allocation delays, NVMe availability delays, and I/O delays.

As expected, we observed linear scaling for most runs, as indicated by the median values and interquartile ranges. However, for runs involving high node counts (7,000 nodes and above), we noticed delays in several executions. These delays could be attributed to various factors, including allocation delays, NVMe availability delays, I/O delays to NVMe, and delays in writing data to Lustre. Notably, the maximum execution time for 9,000 nodes (1.152 million tasks) is 561 seconds, which is significantly less than 10% of the overhead time reported in [7] for a workflow with 100,000 tasks.

GPU Scalability. To validate and verify GNU Parallel's efficiency in launching parallel processes on GPUs, we conducted a weak scaling study using the Celeritas GPU application tasks (refer Section IV for application details). These tasks were run over 10 up to 100 nodes, with each node running 8 processes (corresponding to the 8 schedulable GPUs on the Frontier compute nodes). The results (Fig. 2) demonstrated linear performance with a narrow variance of less than 9 seconds, indicating efficient GPU utilization driven by GNU Parallel.

Stress Tests. We explored the limits of GNU Parallel's capability to handle extreme HT-HPC workflows by determining the maximum process launch rate. This metric is crucial as it sets a lower bound on the size of individual tasks in an HT-HPC workflow. Our findings (Fig. 3) show that a single instance of GNU Parallel can launch approximately 470 processes per second. Using 256 CPU threads on a Perlmutter CPU-only compute node, full utilization is achieved if tasks run for at least 545 milliseconds. Multiple parallel instances of GNU Parallel can increase this rate, with an upper bound of approximately 6,400 processes per second, allowing task duration as short as 40 milliseconds to maintain full node

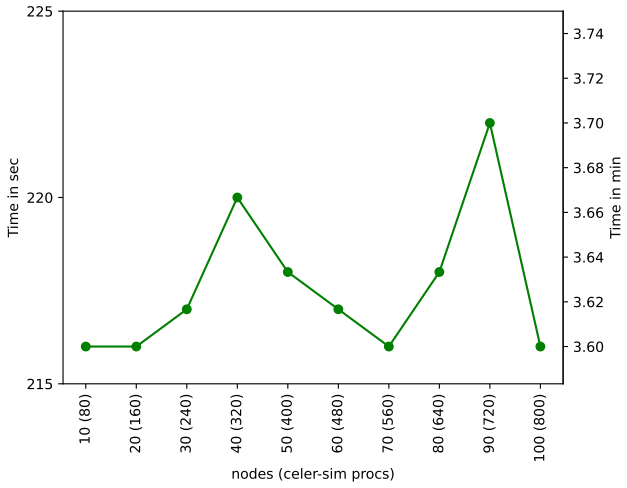

Fig. 2. Weak scaling on Frontier GPU nodes with Celeritas demonstrates linear performance and efficient GPU utilization by GNU Parallel. The variance in execution time was less than 10 seconds across runs on 10 to 100 nodes, each running 8 GPU processes per node.

utilization.

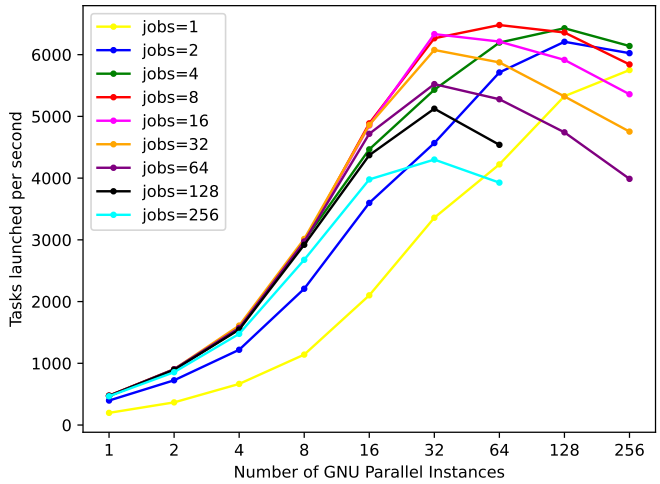

Fig. 3. Maximizing tasks launched per second on Perlmutter with multiple instances of GNU Parallel. A single instance launches 470 processes per second, with full utilization at 545 milliseconds per task. Multiple instances increase this rate to 6,400 processes per second, with tasks as short as 40 milliseconds.

Containers. Stress testing container services using Shifter on Perlmutter CPU nodes yielded promising results, with a container launch rate upper bound of approximately 5,200 processes per second. This indicates a Shifter container startup overhead of only 19% compared to "bare metal" performance, as detailed in Fig. 4. We also performed early experiments with Podman-HPC. However, Podman-HPC's interface is currently less refined than Shifter's, and our tests revealed a significantly lower launch rate upper bound of approximately 65 processes per second (Fig. 5). This performance is two orders of magnitude slower than Shifter, making Podman-HPC less suitable for HT-HPC workloads. Additionally, reliability issues, such as failures in setting user namespaces, database locking, setgid failures, and problems with task tmp directories, were observed at larger scales.

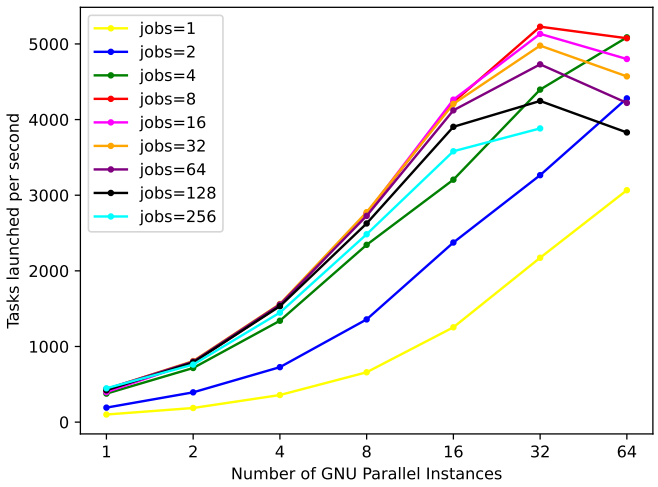

Fig. 4. Maximizing container launches per second on a single Perlmutter CPU node using Shifter. The upper bound was approximately 5,200 processes per second, with a startup overhead of only 19% compared to bare metal performance.

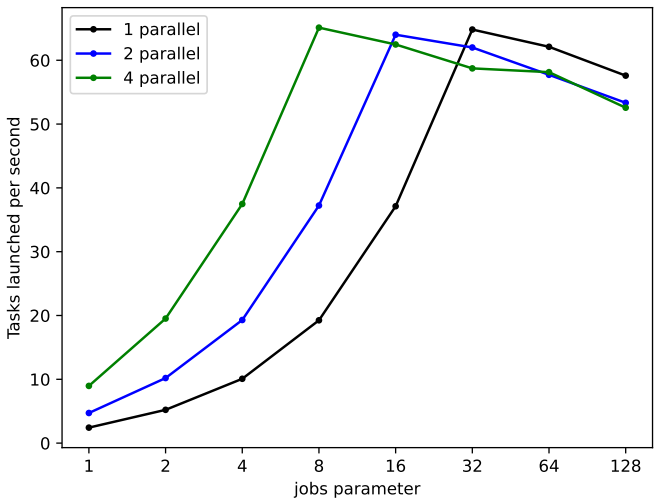

Fig. 5. Maximum Podman-HPC containers launched per second on a Perlmutter CPU node. The upper bound was approximately 65 processes per second, significantly lower than Shifter (see Fig. 4). The jobs parameter is the number of parallel jobs per instance of GNU Parallel set using the `-j` flag.

IV. Applications

In this section, we present several HT-HPC workflow applications that we implemented using GNU Parallel on Frontier, utilizing its Slurm job manager and scheduler. GNU Parallel offers unique advantages when used alongside Slurm and its `srun` directive. First, with a specified `--jobs` parameter, GNU Parallel automatically maintains a pool of additional processes queued in batches according to the `--jobs` setting.

This ensures efficient utilization of available resources and smooth handling of parallel tasks. Second, running multiple instances of GNU Parallel scales and performs significantly better than the `srun` directive alone. This is because `srun` may initially create a resource allocation for each run, and a large number of `srun` invocations can impact the overall scheduler performance. By contrast, GNU Parallel efficiently manages the resource allocation and execution of multiple processes, minimizing the overhead and improving the overall efficiency of high-throughput computing tasks.

A. Motivating Use-Case: Data Fetch-Process Workflow

This motivating example demonstrates how GNU Parallel may be used in instrumenting asynchronous workflow stages, maximizing resource utilization by concurrently performing I/O operations alongside computational tasks or multiple compute stages. This approach allows HT-HPC workflow tasks to run with a synchronization barrier (Fig. 6)[1], ensuring efficient and seamless integration of data fetching and processing.

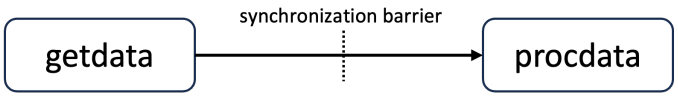

Fig. 6. A canonical representation of two workflow stages with synchronization barrier.

A mock application shown in Listing 2 illustrates a data-fetch→data-process canonical workflow that leverages GNU Parallel to download and process data simultaneously, demonstrating how I/O operations can be interleaved with computation to maintain high resource utilization. The `getdata` script below continuously downloads image data from eight different regions every 30 seconds, utilizing GNU Parallel to perform these downloads concurrently. The timestamp of each download batch is recorded in a queue file (`q.proc`).

```
#!/bin/bash

while true
do
 ts=$(date +%s)
 # download image data from 8 regions
 parallel -j8 "curl -s \
 https://cdn.star.nesdis.noaa.gov/GOES16/ABI/\
 SECTOR/{}/GEOCOLOR/1200x1200.jpg \
 --output ./data/{}_${ts}.jpg" ::: \
 cgl ne nr se sp sr pr pnw
 echo $ts >> q.proc
 sleep 30
done
```

Listing 2. Data fetch-process workflow task using GNU Parallel.

The `procdata` script (Listing 3) reads from the queue file and processes the downloaded images in parallel. It uses the `convert` command to perform image processing tasks, ensuring that computational tasks can run concurrently with data fetching operations. This setup allows for efficient utilization of computational resources, with processing tasks commencing as soon as data becomes available, without waiting for all data fetch operations to complete. This idea of queues as a data exchange may be utilized for larger, production workflows by utilizing centralized message queue systems such as Apache Kafka [15] for heavier data exchange.

```
#!/bin/bash

touch q.proc ; tail -n+0 -f q.proc | \
 parallel -k -j8 'printf "\nTimestamp:{}\n"; \
 convert \
./data/*_{}.jpg -fuzz 10% -fill white \
 -opaque white -fill black +opaque \
 white -format "%[fx:100*mean] " info:'
```

Listing 3. This script reads from the queue file and processes images in parallel ensuring concurrent computation and data fetching for optimized resource utilization.

B. Massive Darshan Log Processing

Darshan is a scalable HPC I/O characterization tool designed to capture and provide comprehensive insights into the I/O behavior of applications running on large-scale HPC systems [16]. Darshan helps researchers and system administrators to identify I/O bottlenecks, optimize filesystem usage, improve application I/O strategies, and make informed decisions about system configuration and resource allocation. The Darshan Massive Log Processing application [17] is designed to handle extensive log data generated over five years. This data requires efficient processing to extract meaningful insights. GNU Parallel significantly enhances this application by reducing the original script size by over 90% and providing automatic queuing for handling input data log files, eliminating the need for the application to manually track them.

Each dataset can be processed independently, fitting the fast node-local NVMe storage. To optimize I/O performance and alleviate the load on the Lustre filesystem, we developed a multi-stage pipeline workflow that mirrors CPU instruction pipelines by prefetching data from Lustre to NVMe in stages. As shown in Fig. 7, the overall workflow consists of five stages: the first stage involves two parallel steps—processing the first dataset from Lustre while simultaneously copying a second dataset from Lustre to NVMe, with the copy operation managed by GNU Parallel-driven rsync processes. Each of the three subsequent stages involves three simultaneous operations: processing from NVMe, copying from Lustre to NVMe, and deleting processed data from NVMe.

The workflow implementation significantly enhances overall performance and reduces time to completion. Specifically, the first stage, which involves processing data directly from Lustre, takes 86 minutes. In contrast, processing data from the faster NVMe storage averages 68 minutes per stage. This approach leads to a total completion time of 358 minutes (86 + (68 × 4)), compared to an estimated 430 minutes

[1]Courtesy of https://tinyurl.com/c9d4pezh

(86 × 5) if all stages were processed solely from Lustre. This represents a 17% improvement in performance. Additionally, by reducing the number of I/O operations on the shared Lustre file system, this method indirectly enhances the overall efficiency of the system, minimizing I/O "hits" and contributing to better resource management.

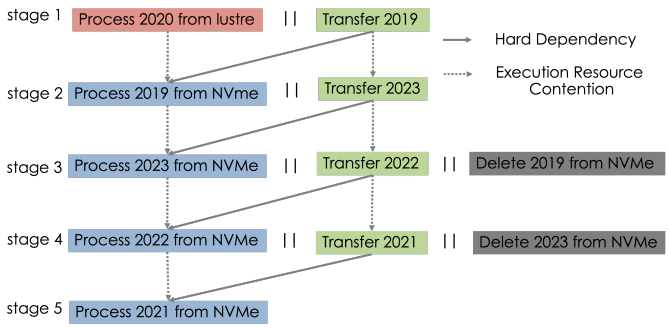

Fig. 7. Darshan Log Processing HT-HPC workflow.

Ease of Use. The script in Listing 4 demonstrates the complexity of the Darshan log processing invocation before the adoption of GNU Parallel. Significant coding and management with `srun` were required to execute the tasks. By contrast, the use of GNU Parallel dramatically simplifies the script, reducing both overhead and complexity, as shown in Listing 5. The implementations of the workflow shown in Fig. 7 are detailed in the artifacts description in the appendix.

```
#SBATCH -N 1
<other SBATCH directives>
module load cray-python
IFS=,
months='1,2,3,4,5,6,7,8,9,10,11,12'
apps_lst='3'

months=($months)
apps_lst=($apps_lst)
counter=0

for month in ${months[@]}; do
 apps=${apps_lst[counter]}
 app=0
 while [[ $app -lt ${apps} ]] ; do
   echo "Month: "${month} " App: " ${app}
   srun -N1 -n1 -c1 --exclusive python3 \
   darshan_arch.py ${month} ${app} &
   sleep 0.2
   ((app++))
 done;
done;
wait
```

Listing 4. Darshan log processing invocation script in bash before GNU Parallel.

```
#SBATCH -N 1
<other SBATCH directives>

module load parallel cray-python
parallel -j36 python3 \
 ./darshan_arch.py:::{1..12}:::{0..2}
```

Listing 5. Darshan log processing invocation script using GNU Parallel.

C. FORGE Large Language Model

FORGE aims to conduct an end-to-end examination of the effectiveness of large language models (LLMs) in scientific research, focusing on their scaling behavior and computational requirements on Frontier [18]. FORGE is a suite of open foundation models that includes models with about 22 billion parameters trained using 257 billion tokens from over 200 million scientific articles. These models exhibit performance that is either on par with or superior to other state-of-the-art comparable models. The effectiveness of FORGE has been demonstrated on scientific downstream tasks, such as using model embeddings for knowledge representation in domain and phase classification, and energy regression. This study provides practical solutions for building and utilizing LLM-based foundation models tailored for scientific research and applications.

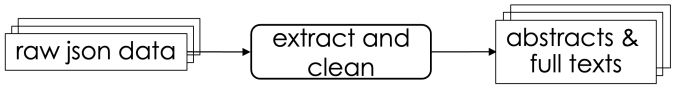

Fig. 8. Preprocessing stage that cleans and curates the raw publications data by extracting abstracts and full texts and removing non-English language and other extraneous characters.

Data curation is a critical step in preparing for effective LLM training, as the quality of the data largely determines the quality of the model itself. GNU Parallel plays an essential role in this process by enabling efficient data cleaning and enrichment, allowing FORGE to handle large volumes of data concurrently. This significantly accelerates the preprocessing phase, ensuring that the models are trained on consistently high-quality data. Moreover, the scripts developed for this workflow are particularly valuable given the frequently updated nature of data sources, such as publication databases. By automating the workflow, these scripts not only save time but also maintain uniform data quality across updates, ultimately enhancing the overall performance and reliability of the LLM.

D. GPU Computing with Celeritas

Celeritas [19], [20] is a new Monte Carlo transport code designed to enhance scientific discovery in high-energy physics by improving detector simulation throughput and energy efficiency using GPUs. Running on Frontier's multi-node GPUs, Celeritas employs a 1-1 process-GPU mapping. Enabling the Celeritas application thus illustrates GNU Parallel's capability to effectively utilize the system's environment with its built-in constructs.

The technique, known as "GPU isolation," maps each GNU Parallel "slot" to a unique GPU. This is achieved by

setting the environment variable `HIP_VISIBLE_DEVICES` (or `CUDA_VISIBLE_DEVICES` for Nvidia GPUs) to the slot number obtained using the {%} built-in construct. For example, the following execution line uniquely distributes the Celeritas process to GPUs:

```
parallel -j8 \
HIP_VISIBLE_DEVICES="$(({%} - 1))" \
celer-sim {} > outdir/{}.out \
::: *.inp.json
```

This approach ensures that each process is assigned to a specific GPU, optimizing the use of available resources. The multi-node execution method can be readily applied here to distribute the processes across multiple nodes, further enhancing the scalability and efficiency of Celeritas simulations.

E. Data Motion

Data staging operations, which involve transferring data from one process to another, are a crucial component of distributed workflows. In this section, we describe how parallel data transfer with low overhead is performed for such workflows. Leveraging `rsync`'s capability to reliably and incrementally transfer data while preserving and creating the necessary directory structure, we have successfully migrated over a petabyte of project data across two parallel file systems. The following example illustrates the use of parallel `rsync` processes for data transfer:

```
find /gpfs/proj/data -type f |\
 parallel -j32 -X rsync -R -Ha {}\
 /lustre/proj/
```

In this method, the `find` command generates a list of files to be transferred, which is then piped to GNU Parallel. GNU Parallel invokes 32 `rsync` processes to perform the data transfer in parallel.

We combined this method with the driver-script based approach described in Section III over an 8-node Slurm-based Data Transfer Node (DTN) cluster. The list of files, generated as standard output, is piped to the driver script, which distributes the files evenly across the DTN nodes. Each node runs an instance of GNU Parallel, invoking 32 `rsync` processes, resulting in a 256-process parallel data transfer operation. This approach significantly enhances the efficiency and speed of data staging operations in distributed workflows—200 speed up over sequential transfers, and over 10 when compared to data transfer protocols used in traditional workflow systems. The measured average transfer throughput was 2,385 Mb/s per node, utilizing 32 `rsync` processes.

V. Conclusion

In this paper, we have demonstrated the significant benefits of using GNU Parallel to manage HT-HPC workflows. Our comprehensive analysis on leading HPC systems such as Frontier and Perlmutter illustrates GNU Parallel's ability to maintain low overhead and high efficiency, even as the scale of computation increases dramatically. By leveraging GNU Parallel, we have optimized CPU, GPU, and NVMe storage utilization, as well as streamlined large-scale data transfer operations.

GNU Parallel's flexibility and ease of use make it a powerful tool for a wide range of applications, from scientific data processing to complex simulation workflows. Its ability to seamlessly integrate with existing HPC environments and workflow management systems provides a robust solution for researchers seeking to maximize resource utilization and minimize orchestration complexity. Our study not only showcases GNU Parallel's scalability and performance but also provides practical insights and methodologies for implementing efficient workflows in extreme-scale computing environments. We encourage the broader adoption of GNU Parallel within the scientific community to enhance the execution of diverse and demanding computational tasks.

In addition to its direct application as a viable workflow manager, GNU Parallel can be used in conjunction with other workflow systems as a ***"last-mile" parallelizing driver*** since its process control and coordination with the underlying system state is far more superior and sophisticated when compared to the workflow management systems that use the programming language provided multiprocessing libraries such as those from Python and Java platforms. Additionally, it may be used as a quick prototyping tool to design and extract parallel profiles from application executions. The potential for GNU Parallel to transform workflow management at extreme scales is substantial, paving the way for more efficient and effective scientific discoveries. In closing, we show that the lowest "time" overhead may be achieved by removing any form of advanced management and just scripting the execution of a workflow, however, we do recognize that it comes at a higher "human" overhead and GNU-Parallel can provide both low time and human overhead.

Acknowledgements

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. This research is partially supported by Laboratory Directed Research and Development Strategic Hire funding No. 11134 from Oak Ridge National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility. We acknowledge Suzanne Parete-Koon, Stefano Tognini, Rick Mohr, Ravi Madduri, and Alexis Rodriguez for their support in various stages of the research and development of this manuscript.

References

[1] R. M. Badia Sala, E. Ayguadé Parra, and J. J. Labarta Mancho, "Workflows for science: A challenge when facing the convergence of HPC and big data," *Supercomputing frontiers and innovations*, vol. 4, no. 1, 2017. [Online]. Available: https://doi.org/10.14529/jsfi170102

[2] R. Ferreira da Silva, R. M. Badia, V. Bala, D. Bard, T. Bremer, I. Buckley, S. Caino-Lores, K. Chard, C. Goble, S. Jha, D. S. Katz, D. Laney, M. Parashar, F. Suter, N. Tyler, T. Uram, I. Altintas *et al.*, "Workflows Community Summit 2022: A Roadmap Revolution," Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2023/2885, 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7750670

[3] O. Yildiz, A. Gueroudji, J. Bigot, B. Raffin, R. M. Badia, and T. Peterka, "Extreme-scale workflows: A perspective from the JLESC international community," *Future Generation Computer Systems*, 2024. [Online]. Available: https://doi-org.ornl.idm.oclc.org/10.1016/j.future.2024.07.041

[4] W. Brewer, A. Gainaru, F. Suter, F. Wang, M. Emani, and S. Jha, "AI-coupled HPC Workflow Applications, Middleware and Performance," *arXiv preprint arXiv:2406.14315*, 2024. [Online]. Available: https://arxiv.org/abs/2406.14315

[5] L. Gombert and F. Suter, "Learning-based approaches to estimate job wait time in htc datacenters," in *Job Scheduling Strategies for Parallel Processing: 24th International Workshop, JSSPP 2021, Virtual Event, May 21, 2021, Revised Selected Papers 24*. Springer, 2021.

[6] "Existing Workflow systems," https://s.apache.org/existing-workflow-systems, 2024.

[7] T. Coleman, H. Casanova, K. Maheshwari, L. Pottier, S. R. Wilkinson, J. Wozniak, F. Suter, M. Shankar, and R. Ferreira da Silva, "Wfbench: Automated generation of scientific workflow benchmarks," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2022. [Online]. Available: http://dx.doi.org/10.1109/PMBS56514.2022.00014

[8] S. Jha, V. R. Pascuzzi, and M. Turilli, "AI-Coupled HPC Workflows," *arXiv preprint arXiv:2208.11745*, 2022. [Online]. Available: https://arxiv.org/abs/2208.11745

[9] O. Tange, "Gnu parallel 20231022 ('al-aqsa deluge')," Oct. 2023, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them. [Online]. Available: https://doi.org/10.5281/zenodo.10035562

[10] L. Casalino, A. C. Dommer, Z. Gaieb, E. P. Barros, T. Sztain, S.-H. Ahn, A. Trifan, A. Brace, A. T. Bogetti, A. Clyde *et al.*, "Ai-driven multiscale simulations illuminate mechanisms of sars-cov-2 spike dynamics," *The International Journal of High Performance Computing Applications*, vol. 35, no. 5, pp. 432–451, 2021. [Online]. Available: https://doi.org/10.1177/10943420211006451

[11] N. Jansson, M. Karp, A. Perez, T. Mukha, Y. Ju, J. Liu, S. Páll, E. Laure, T. Weinkauf, J. Schumacher *et al.*, "Exploring the ultimate regime of turbulent rayleigh–bénard convection through unprecedented spectral-element simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3581784.3627039

[12] R. Mueller-Bady, M. Kappes, L. Atkinson, and I. Medina-Bulo, "Multijob: a framework for efficient distribution of evolutionary algorithms for parameter tuning," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1231–1238. [Online]. Available: https://doi.org/10.1145/3067695.3082476

[13] K. Klenk and R. J. Spiteri, "Improving resource utilization and fault tolerance in large simulations via actors," *Cluster Computing*, pp. 1–18, 2024. [Online]. Available: https://doi.org/10.1007/s10586-024-04318-5

[14] A. Samdani and U. Vetrivel, "Poap: A gnu parallel based multithreaded pipeline of open babel and autodock suite for boosted high throughput virtual screening," *Computational biology and chemistry*, vol. 74, pp. 39–48, 2018. [Online]. Available: https://doi.org/10.1016/j.compbiolchem.2018.02.012

[15] M. J. Sax, *Apache Kafka*. Cham: Springer International Publishing, 2018, pp. 1–8. [Online]. Available: https://doi.org/10.1007/978-3-319-63962-8_196-1

[16] P. Carns, "Darshan," in *High performance parallel I/O*. Chapman and Hall/CRC, 2014, pp. 351–358.

[17] A. M. Karimi, A. Khan, S. Oral, and C. Zimmer, "Summit Darshan Archival Dataset," 2024. [Online]. Available: https://www.osti.gov/biblio/2305496

[18] J. Yin, S. Dash, F. Wang, and M. Shankar, "Forge: Pre-training open foundation models for science," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3581784.3613215

[19] Johnson, Seth R., Esseiva, Julien, Biondo, Elliott, Canal, Philippe, Demarteau, Marcel, Evans, Thomas, Jun, Soon Yung, Lima, Guilherme, Lund, Amanda, Romano, Paul, and Tognini, Stefano C., "Celeritas: Accelerating geant4 with gpus*," *EPJ Web of Conf.*, vol. 295, p. 11005, 2024. [Online]. Available: https://doi.org/10.1051/epjconf/202429511005

[20] S. R. Johnson, A. Lund, S. Y. Jun, S. Tognini, G. Lima, P. Romano, P. Canal, B. Morgan, T. Evans, V. R. Pascuzzi, and D. Deeb, "Celeritas," [Computer Software] https://doi.org/10.11578/dc.20221011.1, jul 2022. [Online]. Available: https://doi.org/10.11578/dc.20221011.1

APPENDIX

ARTIFACT DESCRIPTION

All the codes used in this work are publicly available and hosted on GitHub:

- Main repository: https://github.com/ketancmaheshwari/gnu-parallel-artifacts
- Benchmarks: https://github.com/ketancmaheshwari/gnu-parallel-artifacts/tree/main/benchmarks
- Applications: https://github.com/ketancmaheshwari/gnu-parallel-artifacts/tree/main/applications