Command Lines vs. Requested Resources: How Well Do They Align?

Ben Fulton
Research Technologies
Indiana University
Bloomington, IN, USA
befulton@iu.edu

Abhinav Thota
Research Technologies
Indiana University
Bloomington, IN, USA
athota@iu.edu

Scott Michael
Research Technologies
Indiana University
Bloomington, IN, USA
scamicha@iu.edu

Jefferson Davis
Research Technologies
Indiana University
Bloomington, IN, USA
majdavis@iu.edu

***Abstract*—In the context of high-performance computing, a significant proportion of users do not develop their own code from scratch but rely on existing software packages and libraries. Many of these packages provide a variety of methods for use on multicore, multinode, or large-memory systems. We examine a set of applications that users run on Indiana University supercomputers, and determine for those applications the software parameter settings controlling CPU parallelism, GPU parallelism, and memory usage. We then investigate the common ways users employ these parameters and how effectively the users have made use of available resources. By comparing the command line parameters used (collected via XALT) to the Slurm resource requests we, are able to determine how users take advantage of the resources they request. This informs us on how to provide better example usages for the software on our systems, and informs future software development efforts, guiding the design of more efficient, user-friendly, and adaptable tools that align closely with the specific needs of the HPC community.**

***Index Terms*—High Performance Computing, Monitoring**

I. Introduction

The Research Technologies (RT) division at Indiana University (IU) operates batch-scheduled HPC systems for a wide variety of researchers at the University, from highly sophisticated faculty members running large-scale simulations, to undergraduates unraveling the mysteries of the Linux operating system. From July 2022 through July 2023, three systems were in operation: Carbonate, a 96-node cluster composed of Lenovo nodes; Quartz, a 92-node cluster composed of Gigabyte nodes provided by HPE; and Big Red 200 a 704-node Cray EX.

Many, if not most, users are less interested in carefully analyzing the performance of their job than in getting their workflow to run without error. While resource providers would prefer that the user carefully analyzed their workload to determine the exact resources (memory, CPU, walltime, etc.) required, we find that in practice users quickly settle on a set of parameters that work and then use them for all jobs. For example, by providing the `-mem=0` flag to a job, the user is guaranteed to get all the memory available on a node, which guarantees, in turn, that their job will not run out memory (or some sort of multi-node approach will need to be taken). As the amount of memory and number of cores available on a single node increases, it becomes less and less likely that the node is being used efficiently. Thus, the parameters provided to the job are crucial to the efficiency of the job in using the system resources. However, once the resources are allocated, it is important that the user applications use them efficiently. Many applications provide parameters that allow the user to specify the resources to use, and many have defaults that are potentially less than the maximum available to them. If a user requests 16 cores per task and then runs an application that defaults to a single thread, 15 cores go unused for the duration of that application. Even if the user copies a `num_threads=4` parameter from an example, 12 cores in this instance still go unused. By examining the command lines actually run by the user for well-known applications, we can determine how efficiently the resources requested are being used.

II. Motivation and Design Considerations

There are a number of ways to configure a batch scheduler, such as Slurm [1], that can affect job throughput and how efficiently researchers are able to make use of resources. Over time, the core count and accelerator counts on a single node have continued to increase, introducing the need for subdividing nodes to allow for a single node to provide resources for multiple workloads. We consider three possible modes of operation that have been used at IU, with the understanding that there are many more possible alternatives. A cluster operated in a *job exclusive* mode allows for only one job per node. A little over a decade ago, this was the default deployment for most clusters. For machines that are targeted for a small number of users and applications that are highly parallelized, this can still be a reasonable alternative. A cluster operated in *user exclusive* mode allows for multiple jobs from the same user to occupy a node, but not more than one user per node. This option can help users with high throughput workflows that have many tasks, where each task requires relatively few resources. Rather than putting the onus on the user to orchestrate the workflow under a single job asking for more resources, the user can rely on the scheduler to run many jobs at once. From the resource provider's perspective, there is little risk of one user's workflow negatively impacting another due to node sharing. A cluster operated in *shared* mode allows for multiple jobs from different users to occupy a node. When the Carbonate cluster was commissioned in 2017, it was

the first HPC system to come into production operation as a shared-mode machine. This followed testing and development of the sharing parameters and `cgroup` controls in Slurm on a previous machine. All subsequently deployed machines, including Big Red 200 and Quartz have been operated in shared mode. Where a machine operating in shared mode used to be more of a rarity, as core counts and memory per node have increased over time, and the types of workflows and scientific disciplines served by HPC systems have expanded, it has become increasingly common.

Users run a variety of applications on the IU clusters: highly-tuned traditional supercomputing applications such as LAMMPS [2]; genome assemblers with high-memory requirements like ABySS [3]; TensorFlow [4] based machine learning applications; complex workflow-based applications driven by SnakeMake [5] or NextFlow [6]; as well as single-task, single- and multi-threaded applications and scripts created by users for many purposes. While these applications have an immense variety of goals, each one must request a limited number of resources to run on the system: nodes, cores, memory, and GPUs and perhaps other trackable resources. We recommend that users have an understanding of the resources they require by determining the application's limitations and constraints. A genome assembler might benefit from more memory per transcript while a watershed hydrological model with extra density added will need extra processing power [7], [8]. The user, however, who is focused on completing the assembly or simulation rather than the resources it uses, has no clear understanding of, or even interest in, these requirements. As HPC applications experts, it should fall to us to recommend appropriate settings, but we find that even with a limited set of applications, understanding the requirements well enough to make recommendations is a difficult task. Here we make an attempt to understand a subset of the applications that are run on the system, and the ways users specify the resources they feel are required.

To maximize job throughput and resource efficiency while minimizing scheduling contention among users, accurate setting of resource parameters is critical for a machine operated in shared mode. While the accurate setting of resource parameters is important for machines operated in job-exclusive or user-exclusive modes, accuracy is only required at the node level. There is no concern around intra-node scheduling contention in either of these modes of operation, but for shared-mode operations users will ideally specify the required amount of CPU cores, memory, accelerators, and walltime so that a node can be efficiently packed with jobs. The mode of operation also has implications for the allocation process. Most centers that have an allocations process for computational resources use a single metric for the allocation (e.g. core hours or node hours), with different types of architectures having some conversion factor. At IU, there is not a strictly enforced allocation process; research groups are required to apply for access to a machine, and their job throughput is adjudicated by Slurm's fair share algorithm. The scheduler currently only considers CPU time in the fair share calculation. In both cases, only a single metric is used to determine access to resources, while job parameters are often multidimensional including core hours, memory, and accelerators. When job parameters are poorly specified it can negatively impact the researchers, either by consuming their allocation credits, or by lowering their job priority via fairshare.

At the start of our investigation, we began by profiling the diverse set of software applications installed on IU's clusters, encompassing scientific simulations, machine learning algorithms, and data-intensive workloads. From these, we identified several commonly used applications that take advantage of task parallelism (MPI) and several that use thread parallelism (OpenMP or other threading variants). Threaded applications commonly provide command-line options controlling the number of threads, so, using data from XALT [9], we examined the command lines generated by users to determine patterns in the configuration. For MPI applications we can simply determine the count of tasks actually launched. In either case, we can compare the number of cores actually used by the job to the number requested by the job script.

III. Data Collection

The data used came from three primary sources: XALT, which gathers information as applications are run by users, including the working directory, the path of the executable and command line parameters, details of the runtime and resources utilized, and other items; Slurm, which logs job information to a database including the number of tasks requested and launched, the number of cores requested, and optionally the amount of memory and GPUs requested if provided. In addition, Slurm saves the job script created for each job and the environment variables in force at the time of the launch. Finally, we went over the documentation for each application or application suite to determine the parameters used to request a thread count.

A. XALT Data Collection

For several years, RT has been gathering data on the applications users run on the clusters by means of XALT monitoring. XALT logs a variety of information about the binaries selected by users, including the path of the executable, the exact command line run by each application, and the Slurm job ID associated with the application. XALT also provides information on application suites that consist of multiple applications. At IU, XALT is configured to track applications on both the compute and login nodes.

XALT tracks applications by means of inserting a static library into the executable via the `LD_PRELOAD` function. It is, however, subject to various limitations; the primary one being that logging every binary run by every user would be a prohibitively large task. For this reason, many binaries that are run are not entered into the database. These include: most binaries in `/usr/bin` (such as `ls` and `mv`); most binaries that run for less than a minute; and a variety of applications deemed to be uninteresting or overwhelming, such as lua and lmod, which are required to run the module system.

Some extensions have been provided with XALT to find characteristics of scripting language modules - extensions for R, Python, and Matlab are available. Python supports a mechanism for loading a system-defined package in each run, and the XALT extension leverages this mechanism to log all import statements. Like the application runs, logging every single import would result in an extremely large database and filtering mechanisms are provided. Similarly for R, a custom data gathering package [10] is loaded via the RProfile.site startup file, and loaded packages are gathered.

We pulled the top 100 applications/application suites by number of runs for each of the three clusters. From these, we selected 40 applications for further analysis. As shown in Table I, the 40 applications represented 33% of the overall number of runs logged by XALT, and 89% of the total core hours.

1) Application Suites: XALT's gathering mechanism records the exact path of every application that is logged; this could include exact duplicate applications such as `/usr/bin/more` and `/usr/local/bin/more`. For purposes of reporting, XALT groups these instances into a single set of applications we refer to as an "Application Suite". This concept is extended by a set of configurable regular expressions that can be provided to categorize applications; XALT reports consider each categorization a single application and denotes that by appending an asterisk to the suite title. For example, "rosetta", "rmpisnow", and "minirosetta.mpi.linux" are all collected into the single application suite "Rosetta*". XALT provides a complex database case statement that computes the suite name when reporting results from the database. We use this categorization in our data collection similarly, borrowing the case statement, and denote the suite name as the "AppCode". If a given application does not appear in any regular expression, it is given an AppCode that consists of the name of the binary, without any path information. In addition, we extended the default set of regular expressions to collect various applications popular among Indiana University users; we collect applications belonging to NCBI Blast, Relion, and Orca among others.

2) Sampling: Another method of saving database space is to discard various runs from the database, based on the amount of time the application runs, and a generated random number. At IU, these results are configured with the following rules: an application that runs for less than five minutes is retained with a probability of .0001; an application that runs for five to ten minutes is retained with a probability of .01, and all applications that run for more than ten minutes are retained. The probability used for sampling each run is stored in the database; thus it can be inferred that an application with a probability of .0001 has been run many hundreds of times at a minimum, and if the application appears in the database several times, the number of runs can confidently be assumed to be in the tens of thousands.

Cluster	XALT Runs	Core Hours
Quartz	59.11%	94.07%
Big Red 200	29.2%	51.9%

TABLE I: Percentage of XALT runs and core hours represented by the top 40 applications.

B. Slurm Data Collection

For each application call used in a Slurm job, we pulled information on the resources requested by the user: memory, tasks, GPUs, and CPUs per task. We categorized each job as single task or multiple task and GPU or non-GPU. In addition, we looked at the job script that the user provided to determine if they actually used srun (the Slurm equivalent to mpirun) to take advantage of multiple tasks when requested.

C. Software Application Parameters

After identifying the 40 applications of interest, we sorted them into three categories: SIMD applications, MISD and single-threaded applications, and script runners. An application that supported both MPI and multiple-thread running styles was considered to be a SIMD application. We generally recommend to users that Big Red 200 be used for SIMD applications and Quartz for other applications. IU has a large computational biology department and bioinformatics applications seem to tend towards non-SIMD applications, so the most popular applications run on Quartz consist primarily of bioinformatics applications (Figure 1) .

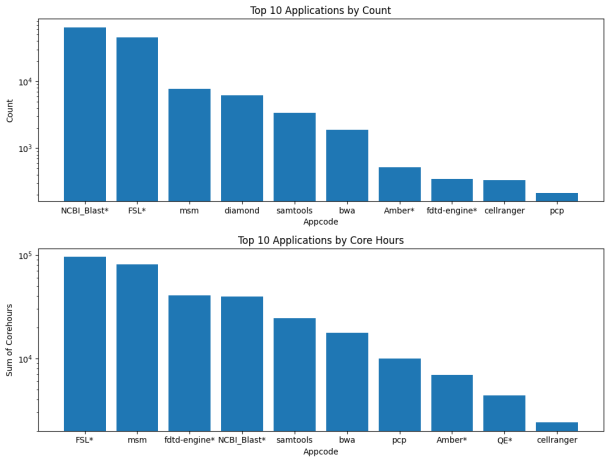

Fig. 1: Quartz Top Applications

The resulting data set consisted of a set of over two million command lines executed by users on the clusters, representing 500,000 distinct jobs and over 50 million core hours (Figure 2) (NB We use XALT's definition of core hours here, which is not the same as those returned by other methods. See Section VI for information on how XALT calculates core hours). Also, for each job, we collected the cores, nodes, memory, and GPUs requested, as well as the setting of `OMP_NUM_THREADS` and the details of any calls to srun or mpirun (Table II).

	Quartz	Big Red 200
Command Lines	1374246	1072780
Job Requests	478886	521179
`OMP_NUM_THREADS`	159071	19167
srun	164245	55556

TABLE II: Collected data size by cluster

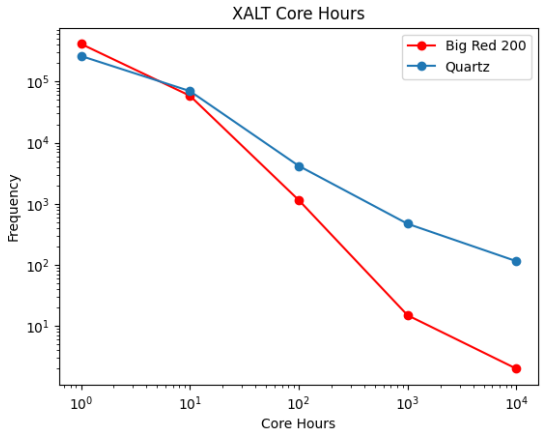

Fig. 2: Core Hours reported by XALT

IV. Results

We performed a variety of analyses on the resulting data set. The results of our study demonstrate the significance of software parameter optimization in maximizing HPC performance. We observe substantial variations in application efficiency depending on parameter configurations, highlighting the need for tailored optimizations for specific workloads. Additionally, the user behavior analysis reveals trends in job submission patterns, which can be leveraged to enhance scheduling strategies and reduce wait times for users.

A. *Overall efficiency of apps*

Of the applications that supported a thread parameter, 5,845 command lines run on Big Red 200 and Quartz actually specified one (Figure 3). Many, if not most, of these runs specified 24 cores; this may be an artifact of users moving from the Carbonate system in which each node had 24 cores. No users took advantage of the full 128 cores of the Quartz system, which seems plausible given the difficulties of efficiently using 128 threads in a single application [11]. Big Red 200, which also provides 128 cores per node, had some less explicable thread count requests, including 5, 30, and 35 threads. If these users were running multiple jobs or multiple nodes per job, several cores per node were going unused.

Of the jobs where a thread count was specified, almost all (99%) requested the correct number of cores via the scheduler to use the threads. In retrospect this is not a surprising result; a user who is sophisticated enough to use multiple threads will generally also know how to request those cores. Users, on the other hand, who called an application that supported a thread count parameter *without* specifying that parameter, had a variety of core requests (Table III). This may indicate a

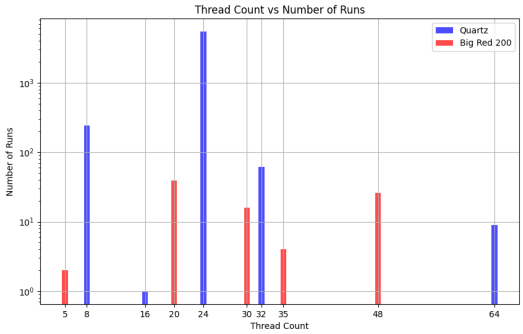

Fig. 3: Command line thread count parameters

lack of knowledge on the users' part of how to effectively run their applications, but there also may be other explanations: running an application that is secondary to the main purpose of the job; intentionally running an application that defaults to using all available threads; running applications in parallel that each require a number of threads; or running an application launched by a script or workflow manager that does not correctly allocate CPUs. Whatever the reason, it seems likely that, at least in some cases, this indicates inefficiently allocated cores.

Requested Cores	Count
1	393
4	1
8	681
24	123
32	4408
48	62
64	12

TABLE III: Requested cores for applications without a thread-count parameter (Quartz)

A similar result was observed when comparing the number of tasks requested in the job script to the number of tasks specified in the srun or mpirun lines (Table IV). Greater than 99% of jobs where both of these numbers were specified matched them correctly; but the majority of jobs did not specify any task count at all. This, again, may point to users who are not clear on how to use all the tasks they've requested; or it may indicate more sophisticated task assignments than we have covered here. (Also, a small number of users specified task counts by using an environment or script variable. We did not attempt to discover the efficiency of these jobs). There did not seem to be any particular pattern in the number of tasks requested when the user did not specify a task count in the srun command (Figure 4).

	Quartz	Big Red 200
Numeric task count	4284	870
Non-Numeric Task Count	16	434
No task count	11413	12410
Matching	4240	814

TABLE IV: Requested tasks specified in srun command lines

Many applications, rather than allowing the user to specify a number of threads, rely on the environment variable `OMP_NUM_THREADS`. When this variable was set, we examined user scripts to determine if the script was explicitly setting the value or if it was set in the environment prior to the script execution. We found it was much more commonly set in the environment, and in this case, it is almost always set to four or fewer threads. It seems likely that this is set as a generic or even accidental parameter, set by the user as a standard part of the environment rather than being intentionally set for a particular run. This may indicate that many users do not understand how Slurm configures the environment for a job. When the value is explicitly set in the script, it is likely to be much greater (Figure 5).

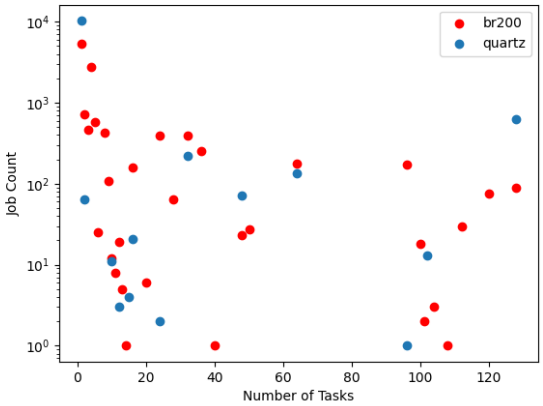

Fig. 4: Requested tasks for jobs with no task specification in an srun call

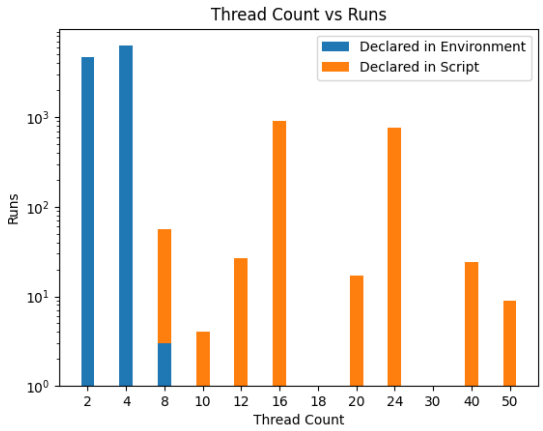

Fig. 5: `OMP_NUM_THREADS` values and source

B. Core Counts by User and by Application

Figures 6 and 7 show a chart of the average count of requested cores by user and by application, respectively. The median of the means for users was 31.03 cores per request, while the median for applications was slightly higher at 46.19 cores per request. There is considerable overlap between these two charts: the users who run applications that support the highest core counts are the users who request the most cores. The higher application median is probably the result of having more users than applications; the wider variety of users may imply the inclusion of newer users reluctant to overburden the system with resource requests. The second and third quartiles ranged from 11 to 64 cores in the case of users; from 23 to 153 cores in the case of applications. PCP is an interesting application case here; averaging more requested cores than any application we studied other than MILC, PCP is a "serial ensemble" job executor [12] rather than a traditional SIMD application.

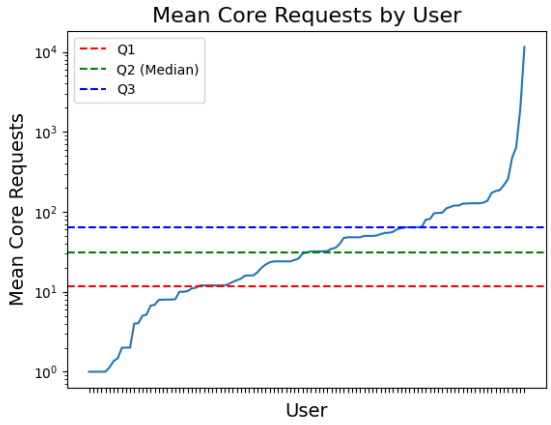

Fig. 6: Mean number of cores requested by individual users

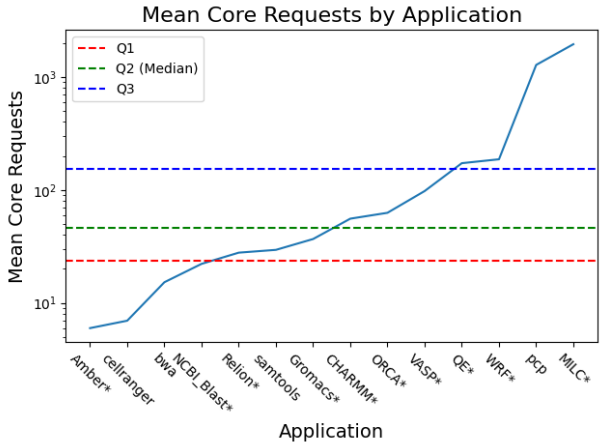

Fig. 7: Mean number of cores requested by various applications

C. Scripting Applications

A confounding factor for our analysis is the large and increasing number of jobs that are driven by scripting languages such as Python, R [13], and Java, where the resource

allocations, if they exist at all, are hidden inside the script. Our database is incomplete in this regard, unfortunately - we have yet to begin R package tracking on Big Red 200 and are not tracking all versions of R on Quartz. However, we did track R versions on Carbonate, a cluster that was retired in 2023. Of the scripting languages we examined, XALT only supports R, Python, and Matlab library loading, and we are not attempting to track Matlab libraries at all.

There are a variety of R packages that can be used for parallelism in R [14]. Historically, the R packages rpvm (R interface to a Parallel Virtual Machine) and Rmpi (interfaces to MPI engines) have been used, but we found no instances of either of these packages in our database. However, more recently, higher-level task-running libraries have been developed such as doParallel and future.

Python is similar. The multiprocessing, threading, and asyncio parallelization packages are part of the base libraries and the packages allow different types of parallelism. In addition, third-party packages such as Dask, MPI4Py, and Joblib provide more sophisticated tooling for users attempting parallel processing. To get an understanding of which of these libraries were commonly loaded by our users, we looked at the relative rankings of these libraries (Figure 8). R packages were commonly in the top 10 libraries loaded. Python parallelization libraries were less popular, probably an artifact of the different collection methods required for each of these languages. The original rpvm package is no longer supported, but we were surprised that no users seem to be attempting to run Rmpi. The newer packages, however, are quite popular with our users and further study of usage patterns of these libraries is warranted. We hope to delve deeper into the threading mechanisms of the more widely used packages and determine how effectively they are used.

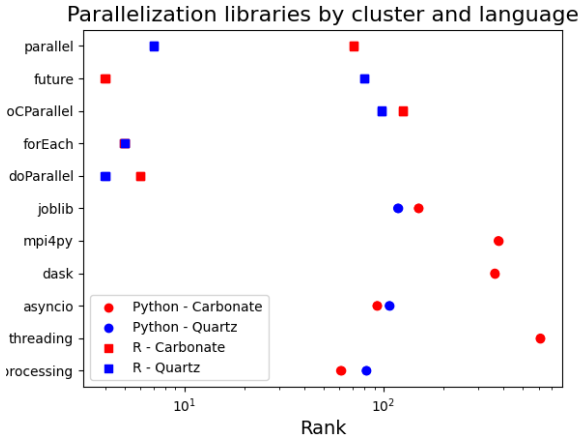

Fig. 8: Popularity of various parallelization libraries. The X axis shows the ranking of the libraries among the most commonly used libraries in that language.

V. Conclusions and Future Work

By using both Slurm logs and data from XALT we were able to gain a deeper understanding of how the HPC systems at IU are being used and gained further insight into the applications being run on the system. For a system that is operated in a shared mode it is critically important that users match their resource requests with the application run parameters to optimize efficient use of the machine and minimize queue wait times for users. These initial results have indicated that, despite our best efforts with outreach and training, there are still a significant number of users that may be over-requesting resources in their Slurm submissions. With this knowledge, we can begin to guide users on requesting and using HPC resources in a way that leads to a more efficient usage of the whole system. As a university, we are training a new generation of students in computational research on a wide variety of applications. It is imperative that we work alongside the research programs to help new students and researchers learn the best ways to utilize HPC systems.

These initial results are indicative of user behavior that we would like to improve, but we have only scratched the surface of analyzing user applications. Going forward, we have two main aims in mind. The first is to extend our analysis and the second is to find the most effective ways to influence user behavior. For the analysis, we plan to include an even wider variety of applications. With clustering and other machine learning techniques we expect to develop a typology of users and user scripts in order to focus our user outreach efforts. We additionally will build on our understanding of user workflows to be able to more accurately detect when users are improperly specifying resources and when they are using the system in an unusual way. We plan to look into using prologue scripts or job submission filtering to prevent or warn users that they may need to re-examine their resource requests to maximize efficiency. Our hope is that with continuous improvement and expansion of the analysis, coupled with more direct user outreach and feedback, we can improve the usage experience for end users and have the university HPC resources be used even more efficiently.

VI. Appendix

Here we provide some details on the data collection methodology.

A. Command Lines

Application command lines are stored in the XALT database as a Binary Large Object, or BLOB. Inside the blob is a JSON string which can be read from the database with the following code:

Listing 1: SQL Command Line

```
select convert(uncompress(cmdline)
  using utf8)
  as cmd from xalt_run;
```

In Python, in turn, this can be converted to a usable object with the JSON module with the simple command

Listing 2: Command Line to JSON

```
j = json.loads(cmd)
```

XALT can also be used to gather srun command lines, but these seem to be a different enough entity that we preferred to pull them directly from Slurm job scripts via the sacct command line. For the time period in question, we requested all job scripts and searched through them to find instances of srun and mpirun.

B. Scripts

Scripts present a special problem in the analysis of software usage. It is rare that the number of threads in use in a script is provided on the command line; rather, if multiple threads are even being used, the number is specified somewhere in the script. Also, many users, as well as many applications released as public software such as Alphafold [15], use scripts as workflow managers, calling various other applications. Since XALT only logs the starting and ending time of each application, a workflow manager written in Python will consider both the application and the Python script to be running for the entire duration of the application, although the amount of processing power used by the script may be minimal. For this reason and others we recommend against attempting to reconcile the core hours reported by XALT to the core hours reported by other logging methods.

In addition, users of XALT may be surprised to not find applications they expect in the database. For example, determining the number of Jupyter runs would seem to be a matter of querying the database for the "jupyter" application, but from XALT's point of view no such application exists. Instead, XALT records that a Python application was run and it included "-m jupyter" on its command line. Jupyter is a script with a shebang header specification telling the system to run it as a Python module.

C. Slurm Job Information

Although Slurm stores its internal information in a database, the provided command-line API can be simpler to work with. For example, reconciling the job ID logged by XALT from an environment variable will not necessarily match directly to the ID stored in the database in a job that was run as part of a job array. This is handled automatically by the API. The API does require a working Slurm instance and some amount of processing and possible network communications, however. Reading the Slurm database directly may be less of a burden on the system.

Acknowledgment

The authors acknowledge the Indiana University Pervasive Technology Institute for providing supercomputing, database, and storage resources that have contributed to the research results reported within this paper.

References

[1] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[2] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.

[3] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "Abyss: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[5] J. Köster and S. Rahmann, "Snakemake—a scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 08 2012. [Online]. Available: https://doi.org/10.1093/bioinformatics/bts480

[6] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature biotechnology*, vol. 35, no. 4, pp. 316–319, 2017.

[7] D. Kleftogiannis, P. Kalnis, and V. B. Bajic, "Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures," *PLoS ONE*, vol. 8, no. 9, p. e75505, Sep. 2013.

[8] Y. Shen and C. Jiang, "A comprehensive review of watershed flood simulation model," *Natural Hazards*, vol. 118, no. 2, pp. 875–902, Jun. 2023.

[9] K. Agrawal, M. R. Fahey, R. McLay, and D. James, "User environment tracking and problem detection with xalt," in *2014 First International Workshop on HPC User Support Tools*, 2014, pp. 32–40.

[10] (2018) Overloaded library and require functions for collecting package usage statistics in r. [Online]. Available: https://github.com/jrmccombs/packagestats

[11] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, Apr. 2012.

[12] A. Thota, S. Michael, S. Doak, and R. Henschel, "Tools to execute an ensemble of serial jobs on a cray," in *Cray Users Group Meeting (CUG2013), Napa Valley, CA*, 2013.

[13] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2021. [Online]. Available: https://www.R-project.org/

[14] (2024) Cran task view: High-performance and parallel computing with r. [Online]. Available: https://cran.r-project.org/web/views/HighPerformanceComputing.html

[15] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, no. 7873, pp. 583–589, Aug. 2021. [Online]. Available: https://www.nature.com/articles/s41586-021-03819-2