ZeroSum: User Space Monitoring of Resource Utilization and Contention on Heterogeneous HPC Systems

Kevin A. Huck
Allen D. Malony
khuck@cs.uoregon.edu
malony@cs.uoregon.edu
OACISS Institute, University of Oregon
Eugene, Oregon, USA

ABSTRACT

Heterogeneous High Performance Computing (HPC) systems are highly specialized, complex, powerful, and expensive systems. Efficient utilization of these systems requires monitoring tools to confirm that users have configured their jobs, workflows, and applications correctly to consume the limited allocations they have been awarded. Historically system monitoring tools are designed for – and only available to – system administrators and facilities personnel to ensure that the system is healthy, utilized, and operating within acceptable parameters. However, there is a demand for user space monitoring capabilities to address the configuration validation and optimization problem. In this paper, we describe a prototype tool, *ZeroSum*, designed to provide user space monitoring of application processes, lightweight processes (threads), and hardware resources on heterogeneous, distributed HPC systems. *ZeroSum* is designed to be used either as a limited-use porting tool or as an always-on monitoring library.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; • **Computing methodologies** → **Parallel computing methodologies**; • **Computer systems organization** → *Multicore architectures.*

KEYWORDS

parallel computing, resource utilization, resource contention, scheduling

ACM Reference Format:
Kevin A. Huck and Allen D. Malony. 2023. *ZeroSum*: User Space Monitoring of Resource Utilization and Contention on Heterogeneous HPC Systems . In *Proceedings of 10th International Workshop on HPC User Support Tools (HUST-23).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3624062.3624145

HUST-23, November 12, 2023, Denver, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3624062.3624145

1 INTRODUCTION

Research on HPC performance tools can broadly be regarded as application-oriented or systems-oriented. Most of the application-oriented tools are concerned with measurement and analysis of an application's performance to identify inefficiencies and opportunities for optimization. In our experience, performance optimization broadly comes down to three types of changes. The first type is wholesale algorithmic replacement, where one implementation is replaced with another that has better theoretical complexity, higher available concurrency, or maps to the available hardware better. Empirical analysis can help to characterize these algorithmic performance problems. The second type is a code transformation change where an algorithm's implementation is modified to improve data locality, reduce data movement, minimize delay, and so on. Finally, there are environmental or configuration changes to how an application is launched and scheduled on a given platform. Without modifying the software, these changes can help increase concurrency, reduce contention, improve utilization, and reduce overall time to solution.

In contrast, systems-oriented performance tools are less concerned about application-specific performance and more focused on monitoring of system status and observing how various components of the system are operating. Practically all HPC systems have some tool support for capturing resource utilization, collecting memory or network actions, recording system call activities, tracking process-level behavior, and so on, as well as interfaces to get access this information during execution. While application-oriented performance tools can access (to the extent possible) systems-oriented monitoring support, it has mostly be in an ad hoc manner.

In this paper we consider that third class of performance optimization – what we will call *configuration optimization* – and the performance technology that could be created to better address the outcomes desired. We suggest that configuration optimization (and/or validation) represents, in fact, "low hanging fruit" that can be automated, but to our knowledge has not yet. However, it involves being able to capture system-side performance information and correlate it with available application performance. Unsurprisingly, this requires monitoring both the system and the application at runtime. Any performance measurement requires some resources to implement. We speculate that application-oriented tools are often reluctant to engage with any degree of monitoring, especially the system. Why? Because it takes additional resources to accomplish (e.g., running background daemons, tracking system calls), potentially limiting application execution and impacting its performance. Our intent is to show that the monitoring we propose is necessary

and useful for configuration optimization, while imposing less than 0.5% overhead. Indeed, we call the tool developed ***ZeroSum*** because its use will help users effectively utilize and distribute the finite set of resources at their disposal without significant cost. The *ZeroSum* prototype is open-source, and freely available on GitHub [1].

2 BACKGROUND AND MOTIVATION

Our interest in configuration optimization is with respect to application performance and we see monitoring as a necessity for a solution. In this context, the concept of *monitoring* includes both real-time access to high-level application performance data as well as monitoring of system status data. For the application, access to the performance data is essential for tasks like run-time optimization, computational steering and other types of adaptive computing. For the system, users would like to know how the system is responding to the demands of the application, how that system data is correlated with available application performance data, and whether there is evidence of low utilization or contention. In this paper, we are mostly focused on the later concept, however we will discuss how the data collected by *ZeroSum* could be integrated into existing performance tools to correlate with application performance data.

In general, we have identified seven different (potentially overlapping) reasons why a user would wish to monitor their application. All have relevance to configuration optimization issues. These reasons include:

- Sanity check
- Check for misconfiguration
- Check for utilization
- Confirmation of expected HW/OS behavior
- Identify cause of failure
- Adaptation
- Identify system failures

Sanity check: Nearly all simulation developers include some kind of logging mechanism, either to standard output / error or to log files. The motivations for this output can include confirmation that the program is making progress, curiosity on what phase/iteration the application is in, or just impatience – not wanting to wait for the application exits to see whether the run experienced a failure or was successful.

Check for misconfiguration: With the popularity and complexity of heterogeneous HPC platforms comes higher complexity with respect to job launch and control systems such as Slurm[43], PBS[14], Alps[28], Torque[19] and Flux[3]. Due to the individual needs of each application, the default – or possibly even the recommended – settings for job launch may be near-optimal for general cases, but they could be sub-optimal for unconventional cases. At a high level, these systems control process placement, logical-physical MPI mappings, resource assignments, and constraints for each process. At a finer detail, environment variables and configuration files control process and light weight process (LWP, a.k.a. thread) placement to sockets, non-uniform memory access (NUMA) domains, cache regions, cores, and hardware threads (HWT). GPU mapping is also performed, and ideally GPUs will be assigned to processes that share a NUMA domain or other local physical connection. The primary concerns for all these assignments involve

[1] https://github.com/khuck/zerosum

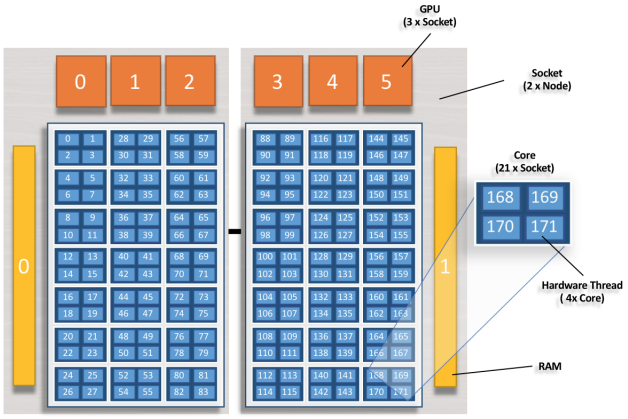

Figure 1: OLCF Summit node diagram [31]. Note that the core ordering in the figure skips from 83 to 88 due to a reserved core for the operating system.

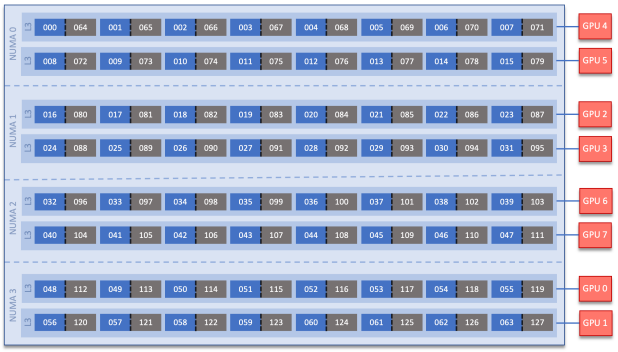

Figure 2: OLCF Frontier simplified node diagram [30]. Note that the GPU indexing [[4, 5], [2, 3], [6, 7], [0, 1]] is non-intuitively associated with the NUMA domain ordering [0, 1, 2, 3].

avoiding both under- and over-subscribing resources and optimizing for data locality. Under-subscription will lead to wasted hardware, energy and allocation time. Over-subscription will lead to increased contention with no realized benefit or worse – longer execution times. Inefficient process mapping can lead to communication latency imbalances between ranks that communicate large data volumes and/or frequently. In summary, because HPC systems are by necessity so flexible and configurable, they are complex and difficult to correctly configure for the average user. Facilities try to provide adequate documentation for each system, but the variability between architectures requires application users to become intimately familiar with the network topologies and node diagrams for each system they use. Whether they *should* be expected to be experts in the systems is debatable.

Figures 1-3 show node diagrams from four different DOE HPC systems, each with varying levels of detail. Even when provided

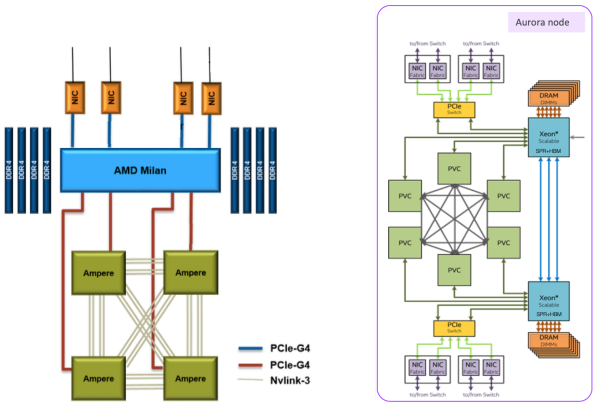

Figure 3: NERSC Perlmutter node diagram [27] (left), and ANL Aurora future node diagram [26] (right). Note that no information is given with respect to GPU ordering, CPU core ordering or how NUMA domains or cache regions are associated with the GPUs (if they are).

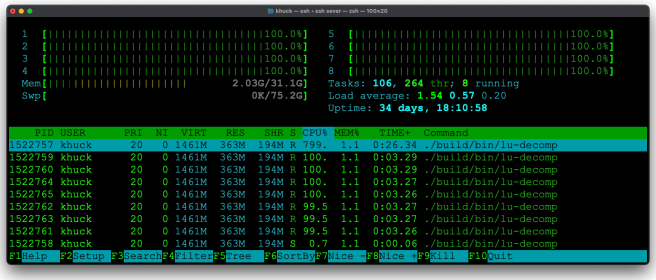

Figure 4: `htop` running on a Linux system. The interface includes information about hardware thread utilization, load, and currently executing processes. Alternative views also show each light weight process (LWP).

information on core indexes, NUMA domains and GPU bus connections, it can be difficult to get an efficient configuration – in particular when the scheduler reserves one or more cores for system processes, and when GPU index, hardware thread, and CPU core index sorting are not intuitive to the user, as in the case of Figure 2 where GPU/GCD 0 is physically connected to NUMA domain 3, containing cores starting with index 48 (or 49 if the first core in the L3 region is reserved by Slurm for system processes).

Check for utilization: Related to the configuration issue, users want to effectively utilize the resources at their disposal. Desktop computer users are familiar with command line utilities such as Linux `top` and `htop` [21, 23] (shown in Figure 4) and `ps`, and even graphical interfaces such as MenuMeters [17], macOS Activity Monitor [5], Windows Resource Monitor [25] and others. HPC facility administrators have access to system monitoring tools like Ganglia [24] and Puppet Console [36], but access to that telemetry data is typically unavailable to users. Users are allocated a fixed number of CPU hours to use, and poorly utilized hardware will lead to wasted allocation time. Depending on whether a given application is CPU- or memory-bound the user may wish to assign more than one operating system thread per core by pinning them to hardware threads. In memory-bound applications, additional contention for an overworked memory interface will lead to delays in execution and longer run times. CPU-bound applications that only schedule one thread per core may be missing an opportunity to better utilize additional hardware threads. GPU accelerated applications may not be fully utilizing the GPUs assigned to each process. The `htop` view in Figure 4 represents a subset of what a user would like to see, but for *all* nodes in a given allocation, and for all resources at their disposal.

Confirmation of expected hardware/software behavior: Kernel monitoring is quite common when it comes to detailed system performance measurement and analysis. The existence of tools like `strace`, `ptrace`, `dtrace`, `dtruss`, `ftrace`, `KUtrace`, `kprobe`, `system-tap`, `eBPF`, `bpftrace`, and many others show robust interest in monitoring what the operating system, kernel and hardware are doing in response to system calls. Unfortunately all of these tools require root level access, kernel modifications/patches, and/or additional processes to monitor the system calls of a given application. Specialized monitoring of subsystems such as Darshan [11] for filesystem I/O provide a window into how well the system is responding to application needs. These are all valuable tools in the hands of system experts, but users may want to know:

- Did I request the "right thing" from the hardware and/or operating system?
- Will I soon run out of a limited resource?
- Do I have alternatives?

For example, Porterfield et al. [34] showed that monitoring the demands on a limited resource (the memory controller) could lead to adaptive thread concurrency control to avoid overtaxing the resource and increasing response latency. While the described solution requires root access to read an off-core hardware counter, there are potentially other situations where monitoring the hardware/OS can help avoid inefficiencies, contention, or failures.

Identify cause of failure: In the event that some unexpected problem (increased latency, reduced throughput, performance variability, contention, application termination) happens, users would obviously like some data on what may have caused the problem. Some of the unexpected but explainable problems include increased or variable network and disk latency, data transfer variability, or resource exhaustion. The exhausted resource could be system queue depths, hardware limits like CPU or GPU physical memory, or file system quotas. Bhatele et al. [6] showed that users cannot avoid "noisy neighbors" (or being one themselves), but that they can mitigate their effects. Mitigating these effects requires some form of monitoring.

Adaptation: Computational steering should not be done blindly, and any attempt to adapt to changing situational changes or evolving data situations requires situational awareness. At a minimum, feedback driven optimizations or control systems requires some observation to drive and confirm the choices. While the monitoring performed by *ZeroSum* likely is not sufficiently detailed or specific enough to support an adaptive system, in some cases it may be useful.

Identify system failures: The authors would like to clearly state that identifying system failures is beyond the scope of this work –

the system administrators at HPC facilities are very good at this already. That said, users would like to confirm that nothing in the user code has caused the failure. Providing the users with a tool that they can use to eliminate other possibilities is useful in reporting actual system failures.

In summary, users would like to know that they have configured their application – with its particular requirements – *correctly* on a given platform – with its particular idiosyncrasies. Prior to any parametric performance study, a performance engineer needs to be confident that they have correctly configured the job launch and scheduling parameters, and are effectively utilizing the available hardware. Before fine-tuning an algorithm for a solver library on a new architecture, a computational scientist or applied mathematician would like to know that they are working with a meaningful use case, and that includes configuring the hardware and operating system correctly.

The main challenges from a tool standpoint are deciding what is going to be captured by the monitoring system (not everything should) and how the application and system will interact to control the monitor operations and manage the data it produces.

3 IMPLEMENTATION

The implementation of *ZeroSum* was inspired by test programs like the `Hello_jsrun` [42] sample program presented in ORNL hackathons and tutorials. Unlike that program, *ZeroSum* is designed to be used with any HPC simulation and detect the process and thread binding behavior of any arbitrary HPC application. We see the evolutional trajectory of a tool like *ZeroSum* to include the following phases:

(1) Detect the initial configuration of the application.
(2) Evaluate the configuration to detect misconfigurations.
(3) Provide runtime feedback to the user that the program is progressing.
(4) Provide a report of how effectively the hardware was utilized.
(5) Provide a report of how much contention was identified in the execution.
(6) Provide a way to export the observed data to other tools that can perform computational steering, if desired.

The design of *ZeroSum* takes into consideration the given requirements for the community of potential users. Those requirements are to provide a generalizable, portable, automatic detection of configuration and evaluate that configuration with a comparison to "known good" configurations. The user needs something akin to warning lights and useful gauges (with explanation), and with very low overhead. The prototype of *ZeroSum* described in this paper includes at least partial support for phases 1, 3, 4, 5, 6.

3.1 Configuration Detection

ZeroSum itself is a C++ library that is injected into an application process space using the standard `LD_PRELOAD` technique of providing the operating system with the path to a library that should be loaded into memory prior to loading the application. That library has multiple ways to initialize itself, either by defining an alternate implementation of the `__libc_start_main()` function – effectively wrapping the `main()` function – or by defining a static global constructor that will be executed when the library is loaded.

Listing 1: Sample hwloc output when executed on a compute node with a single Intel® Core™ i7-1165G7 CPU with four cores and two *processing units* (PU, also known as hardware threads) per core. Note how the logical index (L#) of the HWT/PU is different than the operating system index (P#), leading to potential confusion for the user.

```
HWLOC Node topology:
Machine L#0
  Package L#0
    L3Cache L#0 12MB
      L2Cache L#0 1280KB
        L1Cache L#0 48KB
          Core L#0
            PU L#0 P#0
            PU L#1 P#4
      L2Cache L#1 1280KB
        L1Cache L#1 48KB
          Core L#1
            PU L#2 P#1
            PU L#3 P#5
      L2Cache L#2 1280KB
        L1Cache L#2 48KB
          Core L#2
            PU L#4 P#2
            PU L#5 P#6
      L2Cache L#3 1280KB
        L1Cache L#3 48KB
          Core L#3
            PU L#6 P#3
            PU L#7 P#7
```

ZeroSum provides both options, depending on which method works reliably with a given operating system. During initialization, *ZeroSum* will detect the resources that have been assigned to the process by querying the `/proc/[self|pid]/status` [22] virtual filesystem data to query the cores and/or hardware threads assigned to the process. The `/proc/meminfo` [22] file is parsed to query the data relevant to the memory subsystem such as the total memory available. If the application uses the MPI library, the hostname and global communicator rank and size are queried. A process-specific log file is opened and the initial configuration of the process is written to disk. An asynchronous thread is spawned by *ZeroSum* to perform background tasks (described later this section) and assigned to the last hardware thread assigned to this process by default (this is user configurable). Optionally, a signal handler is installed in order to report a backtrace in the unlikely event of a segmentation violation, bus error, or other abnormal exit. The hardware data model is organized as a tree-based class hierarchy consisting of the compute node, hardware threads (HWT), and GPUs. The software is organized as a tree-based hierarchy of processes and lightweight processes (LWP), commonly referred to as "threads".

If available, the Portable Hardware Locality (hwloc) [10] library is used to query the topology for the available hardware. Hwloc is

integrated with several MPI implementations and job schedulers to assign affinity lists for parallel and distributed applications. [35]. Currently, *ZeroSum* will only utilize hwloc to query and print the hardware topology for the node, similar to the output from the hwloc `lstopo` command. However, for users who may have never used `lstopo` or known of its existence, it is helpful to see how cores are distributed among NUMA domains, which caches are shared between cores, and how HWT are indexed. This information can help the user in choosing an efficient thread placement strategy. In the future, hwloc could be used by *ZeroSum* to automatically (re)assign threads to HWT based on detection of bad configurations. Listing 1 shows sample output from *ZeroSum* when executing on a test system of four cores with two HWT each.

3.1.1 POSIX Threads. The asynchronous thread launched by *ZeroSum* is used to query the state of all the threads in the application. One way to capture thread creation on *POSIX* systems is to wrap or intercept the `pthread_create()` function, however there are ways to circumvent that approach. For example, an application or library might use the `dlsym()` function to query for that symbol, assign it to a function pointer and then call it. As convoluted as that process is, it allows libraries with threaded and non-threaded implementations to co-exist in the same library without imposing a libpthread dependency. Regardless, there is a much simpler method to detect and monitor threads. The `/proc` [22] virtual filesystem provides the `/proc/[self|pid]/task` [22] directory which contains all of the current LWP identifiers for all of the threads in the process. While transient threads (those whose lives are shorter than the periodicity of the asynchronous thread) may be missed, this is a worthwhile trade-off. Even after the thread is created, it may not yet be restricted/assigned to a socket, core, or HWT, so *ZeroSum* still has to periodically query the affinity list of the thread to confirm that it has not changed. In addition to confirming the affinity of the LWPs in the process, *ZeroSum* will also capture the LWP states. This data includes how much time (in the last query period) was spent in user space execution, how much time was spent in system calls, how much time the thread was idle, the current state of the thread (running, idle, blocked, etc.), and how many context switches have occurred, whether voluntary or non-voluntary.

3.1.2 OpenMP Threads. Typically, OpenMP runtimes will launch a team of threads that by default is the same size as the number of threads assigned to the process. For example, if a process is assigned to a range of cores with the Linux `taskset --cpu-list 0-7 <executable>` command, a team launched in a parallel region with default settings will contain eight total threads. For OpenMP runtimes prior to the 5.1 standard [32], *ZeroSum* will launch a parallel region with the `#pragma omp parallel` pragma, and get the thread indices and LWP identifiers for the threads in the team of threads (that typically live for the duration of the application, or until the OpenMP runtime is exited). However, this technique is not reliable for all runtimes or all situations, for example when the `#pragma omp teams num_teams(N)` pragma is used in the application. A more reliable method that *ZeroSum* also supports is to use the OpenMP Tools (OMPT) support in modern runtimes (OpenMP standard version 5.1 compliant or newer) to request that the runtime notify the tool with a registered callback when an OpenMP thread is created. *ZeroSum* has a minimal integration with OMPT that includes support for this callback when provided by the OpenMP runtime. Using the callback, *ZeroSum* can identify the underlying *POSIX* thread as supporting an OpenMP thread. Because OMPT integration requires a 5.1 or newer compliant runtime, the first method described is provided for runtimes such as the one provided by GNU and support is automatically detected at configuration time.

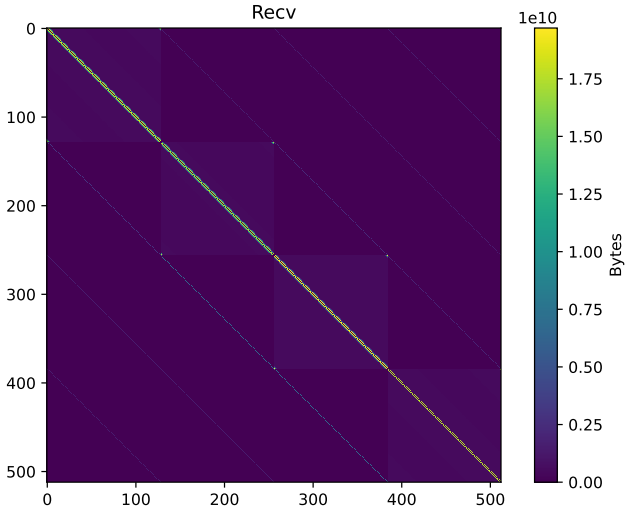

Figure 5: MPI point-to-point heatmap data of a gyrokinetic particle-in-cell code [16] launched with 512 ranks running on Frontier, showing a strong nearest-neighbor pattern along the central diagonal.

3.1.3 MPI. As mentioned, during startup the hostname and MPI communicator rank and size are captured by the asynchronous thread, after it detects that MPI has been initialized through the `MPI_Initialized()` call. *ZeroSum* also wraps the MPI point-to-point API calls to capture the total bytes transferred and the rank of the sender and receiver. This data helps give the user a high level understanding of which MPI ranks are communicating the most frequently with each other. This data can be post-processed to generate communication heatmaps like the one shown in Figure 5. This data could also be used to guide the logical MPI process ordering on the nodes to exploit lower latency communication between ranks executing on the same node.

3.2 Configuration Evaluation

ZeroSum does not yet have any capability to detect and report a misconfiguration. As mentioned earlier, a diagnosis would require an evaluation of the existing configuration as well as a comparison to a known "good" configuration for this platform. The authors see this as a useful yet potentially time-consuming process of determining good configurations on specific platforms of note. However, there are some easy benefits available in automatically detecting when one or more LWPs are assigned to the same set of HWTs, and there is a detectable contention between them. Because *ZeroSum* is in an early prototype stage, this capability has not yet been added.

3.3 Progress Detection

At a very basic level, *ZeroSum* has the ability to periodically write data to `stdout` indicating that at a minimum, the application is viable. While this is the extent of the current functionality in *ZeroSum* to provide a positive "heartbeat" to the user, we observe other opportunities for reporting progress or detecting deadlock. Provided the asynchronous periodic thread has not been blocked, *ZeroSum* should be able to evaluate the status data available for each thread. For example, the LWP and HWT idle/user/system counters are still available, as is the current state of the LWP. There is also an opportunity to examine the current instruction pointer for each thread, and between all of those inputs, detect a deadlock condition and possibly terminate the application to prevent wasting of allocation resources. We see this as an interesting future research opportunity. In addition, *ZeroSum* could potentially be integrated with data services, providing a continuous stream of data reporting the current state of the application.

3.4 Utilization Report

At the end of execution, a utilization report is generated. Rank 0 will write a summary report to `stdout`, while all ranks will write a detailed report to their log files. First, the duration of the execution is reported, along with basic information about the process including the MPI rank, the process ID, the hostname of the node, and the process affinity list. Then a list of all LWP detected during execution is reported as a table. The table includes the LWP ID, the thread type (Main, ZeroSum, OpenMP, other), the percentage of time spent in system calls, the percentage of time spent in user code, the number of non-voluntary context switches, the number of voluntary context switches, and the affinity list of HWT indexes that the thread was assigned. It should be noted that some threads, like MPI or GPU progress/helper threads are not restricted to any set of cores (and are typically not bound by job schedulers). The LWP system percentage added to the user percentage should add up to 100%, indicating that the LWP was not sitting idle. Whether that time was "useful" is beyond our scope, as code efficiency is better measured by existing detailed performance tools looking at hardware counters. Listing 2 shows example output from an MPI execution of miniQMC [18, 37] with 8 processes, 4 OpenMP threads per process and GPU target offload. The number of threads controls the number of walkers in the algorithm, each of which executes target offload calls to a single AMD MI250X GCD, or one half of the full GPU.

After the LWP report, the HWT report is listed. The HWT report is limited to the HWTs that are part of the affinity list of the process, although some threads, like the MPI progress thread, are not bound to any affinity list. The HWT report includes the index, the percentage of time that the HWT was idle, the percentage of time the HWT spent in system calls, and the percentage of time the HWT spent executing user code. Other metrics (nice, idle, iowait, etc.) are also available but not currently captured. The output in Listing 2 shows that half of the cores were idle, and those that were executing spent a notable amount of time in system calls, likely as part of the target offload data transfers, kernel launch, and synchronization process. The idle time is likely due to synchronization waiting on the GPU to complete the requests.

In addition to the CPU utilization data, the GPU utilization is also shown. The "visible" HIP index (0) of the GCD/GPU is shown, even though the true GCD/GPU index (4) may be different. The data shown in Listing 2 is collected using the ROCm SMI API [4]. For other architectures (CUDA, SYCL), *ZeroSum* is integrated with the NVIDIA NVML library [29] and Intel DPC++/SYCL API [12] to query similar statistics. In the summary view the minimum, mean, and maximum observed values are shown.

3.5 Contention Report

The data reported in the Utilization Report is useful in understanding the amount of contention that a LWP experienced during execution. One obvious metric is that the the non-voluntary context switches measure how frequently the operating system had to interrupt the LWP in order to provide time-sliced access to the HWT that the LWP is running on. The percentage of time spent in system calls can intuitively give the user some indication of how much contention the thread experienced, as system calls are used by the application to utilize limited system resources such as network, I/O, semaphores and mutexes. Comparing the affinity list for a given LWP with the other LWPs in the process can indicate whether there was any overlap in resource assignments – which will usually introduce non-voluntary context switches to the LWP report due to the over-subscription of the HWT/core.

In addition to thread contention, there are other finite resources to monitor. One such resource is memory usage, both the overall system memory as well as the GPU memory (if present). For the system memory, *ZeroSum* monitors the `/proc/meminfo` [22] file as well as the `/proc/[self|pid]/status` [22] file to check both how much total system memory is available as well as how much each process is using. This helps to capture whether an out-of-memory error (OOM) has happened due to the application processes themselves or due to another system process that is consuming large amounts of memory. On the GPU, *ZeroSum* will use the vendor libraries listed in Section 3.4 to periodically check how much memory is used and free on the GPU resources.

3.6 Exportation of Data

Each process monitored by *ZeroSum* will write a log file containing the same summary that was written by the root process to `stdout`. In addition to the high level summary, a detailed dump of all data collected by *ZeroSum* is also written to the log as comma separated values, allowing for time-series analysis of the periodic data. The LWP CSV values include additional data such as the current state of the thread (running, sleeping, waiting, etc.), minor page faults, major page faults, the number of pages swapped, and the CPU number that the LWP was last executed on. The log file also contains the MPI point-to-point data collected between all ranks, which can be post-processed to produce a heatmap like the one shown in Figure 5.

4 EXPERIMENTAL EVALUATION

To date, *ZeroSum* has been tested on the Oak Ridge National Laboratory machines Summit [31] and Frontier [30], as well as the the NERSC machine Perlmutter [27] and an internal test system with an Intel Xe GPU accelerator at *(hidden for anonymous submission)*.

Listing 2: Sample output from miniQMC (the OpenMP target offload implementation) executed on Frontier, using the OpenMP environment variables `OMP_PROC_BIND=spread`, `OMP_PLACES=cores`, `OMP_NUM_THREADS=4`, and launched with `srun -n8 --gpus-per-task=1 --cpus-per-task=7 --gpu-bind=closest`. The first core from each NUMA domain was reserved for system processes, and only one HWT per thread was enabled using the `#SBATCH --threads-per-core=1` argument to the job script.

```
Duration of execution: 210.878 s

Process Summary:
MPI 000 - PID 51334 - Node frontier09085 - CPUs allowed: [1,2,3,4,5,6,7]

LWP (thread) Summary:
LWP 51334: Main,OpenMP - stime: 12.48, utime: 63.94, nv_ctx:   4, ctx: 365488, CPUs: [1]
LWP 51343:     ZeroSum - stime:  0.15, utime:  0.26, nv_ctx:   9, ctx:    679, CPUs: [7]
LWP 51374:       Other - stime:  0.00, utime:  0.00, nv_ctx:   0, ctx:      6, CPUs:
     [1-7,9-15,17-23,25-31,33-39,41-47,49-55,57-63,65-71,73-79,81-87,89-95,97-103,
      105-111,113-119,121-127]
LWP 51384:      OpenMP - stime: 12.60, utime: 64.00, nv_ctx:   3, ctx: 365742, CPUs: [3]
LWP 51385:      OpenMP - stime: 12.63, utime: 64.27, nv_ctx:   2, ctx: 352574, CPUs: [5]
LWP 51386:      OpenMP - stime: 12.74, utime: 63.76, nv_ctx: 473, ctx: 368585, CPUs: [7]

Hardware Summary:
CPU 001 - idle:  22.70, system:  12.42, user:  64.52
CPU 002 - idle:  99.82, system:   0.00, user:   0.00
CPU 003 - idle:  23.08, system:  12.60, user:  63.97
CPU 004 - idle:  99.83, system:   0.00, user:   0.00
CPU 005 - idle:  22.79, system:  12.62, user:  64.23
CPU 006 - idle:  99.83, system:   0.00, user:   0.00
CPU 007 - idle:  22.94, system:  12.89, user:  63.81

GPU 0 - (metric: min  avg  max)
     Clock Frequency, GLX (MHz): 800.000000 1614.691943 1700.000000
     Clock Frequency, SOC (MHz): 1090.000000 1090.000000 1090.000000
     Device Busy %: 0.000000 14.616114 52.000000
     Energy Average (J): 0.000000 8.328571 10.000000
     GFX Activity: 0.000000 17223.704762 38443.000000
     GFX Activity %: 0.000000 13.706161 41.000000
     Memory Activity: 0.000000 623.623810 1536.000000
     Memory Busy %: 0.000000 0.355450 3.000000
     Memory Controller Activity: 0.000000 0.303318 2.000000
     Power Average (W): 90.000000 126.483412 138.000000
     Temperature (C): 35.000000 37.909953 39.000000
     UVD|VCN Activity: 0.000000 0.000000 0.000000
     Used GTT Bytes: 11624448.000000 11624448.000000 11624448.000000
     Used VRAM Bytes: 15044608.000000 4743346651.601895 4839596032.000000
     Used Visible VRAM Bytes: 15044608.000000 4743346884.549763 4839596032.000000
     Voltage (mV): 806.000000 891.848341 906.000000
```

This collection of machines covers several CPU and GPU architectures, as well as different job schedulers. For all of the platforms, if hwloc is available configuration and building of the tool requires that `PKG_CONFIG_PATH` has been set to the path for hwloc.

For brevity, we focus on results from the Frontier system. Frontier is a HPE Cray EX supercomputer located at the Oak Ridge Leadership Computing Facility. The system has 74 Olympus rack HPE cabinets, each with 128 AMD compute nodes, for a total of 9,408 AMD compute nodes. Each compute node consists of one

LWP	Type	stime	utime	nvctx	ctx	CPUs
18351	Main†	1.54	15.17	332905	1838	1
18356	*ZeroSum*	0.42	1.10	194	1007	1
18385	Other	0.00	0.00	0	41	1-127‡
18405	OpenMP	0.31	13.09	232689	5	1
18407	OpenMP	0.44	12.93	353365	11	1
18408	OpenMP	0.21	13.22	92528	3	1
18409	OpenMP	0.47	12.93	394014	10	1
18410	OpenMP	0.37	13.03	302371	7	1
18411	OpenMP	0.41	12.97	348829	10	1

Table 1: Frontier results, default configuration. †indicates that the main thread is also an OpenMP thread. ‡indicates that the first core of each L3 region was set aside for system processes, not all threads in the sequence 1-127 are allowed but summarized for brevity in the table (see LWP 51274 in Listing 2).

LWP	Type	stime	utime	nvctx	ctx	CPUs
18552	Main†	3.13	88.40	5	704	1-7
18561	*ZeroSum*	0.79	2.64	2	2790	7
18588	Other	0.00	0.00	0	41	1-127‡
18589	OpenMP	1.10	90.00	9	716	1-7
18590	OpenMP	1.10	93.00	8	724	1-7
18591	OpenMP	1.07	90.52	9	692	1-7
18592	OpenMP	1.10	89.83	14	766	1-7
18593	OpenMP	1.10	90.48	7	728	1-7
18594	OpenMP	1.10	91.93	300	849	1-7

Table 2: Frontier results, configuration requesting 7 cores per process. †indicates that the main thread is also an OpenMP thread. ‡indicates that the first core of each L3 region was set aside for system processes, not all threads in the sequence 1-127 are allowed but summarized for brevity in the table (see LWP 51274 in Listing 2).

64-core AMD "Optimized 3rd Gen EPYC" CPU, 2 hardware threads per physical core, and access to 512 GB of DDR4 memory. Each node also contains four AMD MI250X, each with two Graphics Compute Dies (GCDs) for a total of eight GCDs per node.

Building *ZeroSum* on Frontier is fairly straightforward, using CMake to configure and build the source code. Library dependencies like HIP/ROCm and hwloc are optional, so if they are not found the tool will still configure and build.

To demonstrate *ZeroSum*, we compile and run the ECP Proxy Application miniQMC [37]. miniQMC contains a simplified but computationally accurate implementation of the real space quantum Monte Carlo algorithms implemented in the full production QMCPACK application.

In the following examples, the CPU-only MPI+OpenMP implementation of miniQMC was compiled with the AMD 5.2.0 compiler using the Cray MPI wrappers and the AMD OpenMP runtime. The problem size executed was [2, 2, 2]. As a simple test, the application was executed with eight MPI ranks, and seven OpenMP threads per process, set with `OMP_NUM_THREADS=7`. The application was launched with the default settings of `srun` and minimal arguments. The default configuration on Frontier is to reserve the first core of each NUMA domain for system processes. The complete command line was `srun -n8 zerosum-mpi miniqmc`. *ZeroSum* was configured to collect samples at a frequency of once per second. Some of the summarized output from *ZeroSum* is shown in Table 1. The table shows that the default behavior is to only allow 1 core per MPI process, and all of the threads were bound to the first available core, core 1. The lone exception is the MPI helper thread, which is not bound to any cores – not even the subset assigned to the process. `Stime` and `utime` are the average time spent in system and user time, measured in jiffies. The `nvctx` column shows the total number of non-voluntary context switches experienced by each thread, and clearly there are a very large number of context switches as the OS is time-slicing the core to share across the 9 total threads. The application reported execution time was 63.67 seconds.

In contrast, the data shown in Table 2 shows what happens when the slurm command is changed to `srun -n8 -c7 zerosum-mpi miniqmc`. In this case, the OpenMP threads are not bound to specific cores, but the OS will schedule them on a core and they typically will not need to be migrated during the short execution of this proxy. In this example execution, the OpenMP threads were all migrated at least once during execution, as captured by *ZeroSum* recording the core on which the thread last executed at each periodic measurement (not shown). The contention for time on each core is reduced significantly for all application threads. One thread that still has some measurable contention is the last OpenMP thread which shares a core with the *ZeroSum* thread that is periodically observing the system. The core/thread where the *ZeroSum* thread executes is runtime configurable with an option passed to the `zerosum-mpi` wrapper script. The application reported execution time was reduced to 27.33 seconds. While this is a simple and obvious change to using the node resources correctly, it helps demonstrate the ability of *ZeroSum* to identify poor thread-core mappings with minimal effort.

Finally, the data shown in Table 3 shows what happens when the slurm command is changed to `srun -n8 -c7 zerosum-mpi miniqmc` and the OpenMP `OMP_PROC_BIND` and `OMP_PLACES` environment variables are set to `spread` and `cores`, respectively. In this case, the OpenMP threads are bound to specific cores. The contention for time on each core is reduced further (threads are never migrated), and the voluntary context switches are also reduced. While the runtime for this example is similar to the second case (27.40 seconds), for a larger, longer running application this configuration could be preferable and therefore would show a measurable performance difference.

Figures 6 and 7 show a time series stacked line chart of the idle, system, and user times recorded by *ZeroSum* for the LWP and HWT, respectively, for the third example described above. These charts demonstrate the potential for producing time series charts of all the data captured by *ZeroSum*, including major and minor page faults, the number of pages swapped, voluntary and non-voluntary context switches, the processor a LWP ran on, and the state of the LWP. It should be noted that Figure 6 is rather noisy, reflecting the fact that `/proc/[self|pid]/stat` [22] data is not accurate enough for detailed performance measurement but is accurate in

LWP	Type	stime	utime	nvctx	ctx	CPUs
18948	Main†	3.07	88.57	2	386	1
18954	*ZeroSum*	0.71	2.57	2	291	7
18981	Other	0.00	0.00	0	41	1-127‡
18992	OpenMP	1.18	96.36	0	422	2
18993	OpenMP	1.14	96.50	1	391	3
18994	OpenMP	1.18	96.46	0	381	4
18995	OpenMP	1.11	93.89	0	324	5
18996	OpenMP	1.14	93.29	0	370	6
18997	OpenMP	1.14	95.54	208	358	7

Table 3: Frontier results, configuration requesting 7 cores per process and binding OpenMP threads to cores. †indicates that the main thread is also an OpenMP thread. ‡indicates that the first core of each L3 region was set aside for system processes, not all threads in the sequence 1-127 are allowed but summarized for brevity in the table (see LWP 51274 in Listing 2).

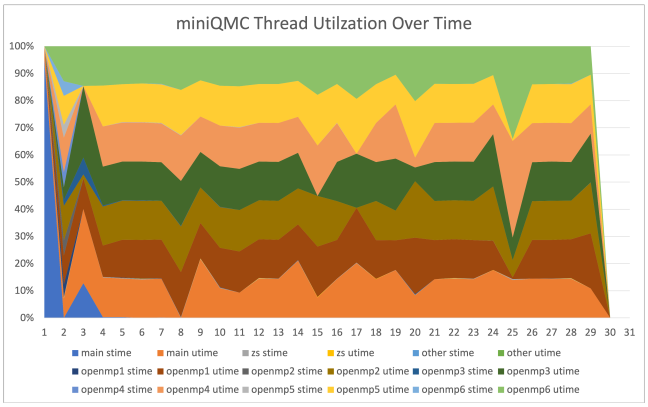

Figure 6: miniQMC LWP (threads) over time. The noisiness demonstrates the lack of precision when using the virtual filesystem to observe utilization data.

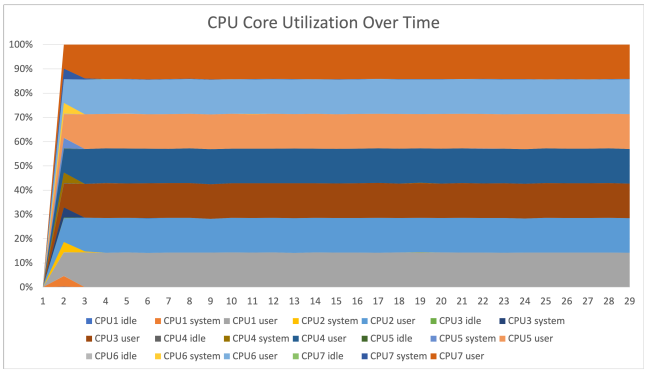

Figure 7: CPU Core utilization over time.

the aggregate and sufficient to meet the motivations outlined in Section 2.

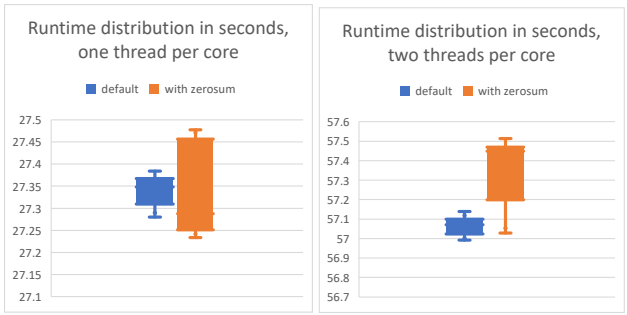

Figure 8: miniQMC time distributions executed 10 times using one OpenMP thread per core (left). In this comparison, the distribution of times with *ZeroSum* is noisier, but there is no significant observation of measurable overhead. The right figure shows the time distributions using two OpenMP threads per core. In this comparison, the distribution of times with *ZeroSum* is both noisier and longer tailed, and does show an observation of overhead, averaging about 0.2752 seconds, or 0.5%.

4.1 *ZeroSum* Overhead

To measure the overhead of *ZeroSum*, we executed the same MPI+OpenMP implementation of miniQMC with the best running slurm configuration (reported earlier) ten times with and without *ZeroSum* in the same allocation on Frontier. We also repeated the experiment using two threads per core, or 14 total OpenMP threads (reserving the first core of each L3 region for system processes as is the recommended and default configuration). As with other experiments, *ZeroSum* was configured to collect samples at a frequency of once per second. For the first comparison shown in Figure 8, we observe no statistically significant difference between the baseline and with *ZeroSum* when comparing the application self-reported runtime. The average runtime was 27.3396 ± 0.0358 seconds in the baseline case, compared to 27.3395±0.1043 seconds with *ZeroSum*. The t-test score comparing the two distributions is 0.998, suggesting a high probability they were drawn from the same distribution. However, the second scenario with two threads per core did show a slight increase in runtime.The baseline case had an observed runtime of 57.0657 ± 0.0486 seconds and the *ZeroSum* case had an observed runtime of 57.3409 ± 0.1823 seconds, with a t-test score of 0.0006 suggesting it is very unlikely these two sets were drawn from the same distribution. However, the observed overhead in this scenario was an average of 0.2752 ± 0.1891 seconds, or less than 0.5%. We suspect the difference between these two cases suggests that when a core is already fully occupied with two busy OpenMP threads, the introduction of the *ZeroSum* thread activity even at 1 second intervals was enough to slightly perturb the overall runtime.

5 RELATED WORK

Much of the work related to *ZeroSum* has already been discussed in Section 2. Command line utilities such as Linux `top` and `htop` [21, 23] (shown in Figure 4) and `ps`, and graphical interfaces such as MenuMeters [17], macOS Activity Monitor [5], Windows Resource Monitor [25] and others are all examples of single-node methods

for observing system utilization and contention. *ZeroSum* is accessing similar user-space interfaces to the reported system data, but aggregating it over all of the nodes in an HPC allocation.

Most system monitoring tools in the HPC and Cloud communities are designed for use by system administrators and that data is rarely shared with users. Examples include LDMS [2], Ganglia [24], Puppet Console [36]. and Jobstats [33]. These approaches all require pre-installed databases, services, and/or daemons to collect, aggregate and analyze the data from all jobs scheduled on the system. Another administrator-focused project is TACC Stats [13]. Due to the volume of data coming from every node in a cluster, the default data collection periodicity of TACC Stats is 10 minutes (scheduled as a cronjob), compared to the 1 second default of *ZeroSum*, although they are both configurable. While TACC Stats collects hardware counters and data at the core level, they do not track thread utilization or contention data from each process. Unlike *ZeroSum*, data does not appear to be immediately available but is aggregated every 24 hours to a central repository for analysis.

Linux system tracing tools such as `strace`, `ptrace`, `dtrace`, `dtruss`, `ftrace`, `KUtrace`, `kprobe`, `system-tap`, `eBPF`, `bpftrace` monitor what the operating system, kernel and hardware are doing in response to system calls. As mentioned in Section 2, these tools require root level access, kernel modifications/patches, or additional processes to monitor the system calls of a given application. They also perform very detailed tracing of system calls, potentially introducing significant overhead and generating volumes of data. These tools are not appropriate for user-space monitoring of distributed HPC applications.

Specialized monitoring of subsystems such as Darshan [11] for filesystem I/O provide a window into how well the system is responding to application needs, and while it provides users access to their utilization data for analysis and visualization, the measurement is limited to filesystem performance.

Several application performance libraries and tools provide access to some monitoring data. PAPI [41] has many components that are designed to observe system resources such as network, filesystem and hardware counters, and performance measurement tools like Score-P [20], Caliper [7], HPCToolkit [1], and TAU [40] utilize PAPI for measurement, but these tools are better suited to targeted application performance analysis scenarios, not user-space monitoring of system utilization.

6 CONCLUSION AND FUTURE WORK

The *ZeroSum* research prototype described in this paper was motivated by a gap that we believe exists in present HPC performance tools between application-oriented and system-oriented environments and the solutions available to address optimization problems of interest to the user. In particular, when considering the *configuration optimization* problem, observing application performance data or system performance data alone is insufficient. Rather, it is necessary to monitor both in order to have a joint context for identifying the complex placement, utilization, and contention performance issues that occur at the application-system boundary.

We developed *ZeroSum* to capture significant information of interest to how the application is scheduled, assigned resources, and executes – information that is available at the system level, but from different sources and about heterogeneous devices that are not commonly integrated in application performance tools. *ZeroSum* is doing the difficult work of tapping into these sources, consolidating the data, analyzing configurability issues, and producing results. It is intended to function as a window on system operation that is integrated with the application and is knowledgeable of the HPC configuration environment.

There are several future directions that are worthwhile to pursue. Our choice of what to include in *ZeroSum* was directed by potential use cases and the system utilities reasonably available to us. As the systems and hardware evolve, it well may be that other data is important to capture for addressing configurability optimization concerns. We will err on the side of utility, only collecting data for which there is an analysis requirement.

Leveraging *ZeroSum* as a "use once when porting an application to a new system or job scheduler" is certainly a motivating use case in our opinion. However, for the purposes of fully integrated user-space monitoring, *ZeroSum* could be extended in several ways. Better connecting to *ZeroSum* during execution is a common interest that can serve multiple purposes. For instance, we can imagine *ZeroSum* being utilized to feed application-oriented information to system-oriented services such as LDMS. Going the other way, interfaces to *ZeroSum* could make its data accessible to application performance tools like TAU. Caliper [8] or PerfStubs [9] would be a good candidate for this purpose. The goal of adapting an application during execution will require even more support for collecting *ZeroSum* data from across the application processes. More sophisticated monitoring infrastructure, such as based on distributed services like Mochi [38], could be pursued. *ZeroSum* could be integrated into runtime systems like Argobots [39] to help guide runtime control decisions. Finally, the log output from *ZeroSum* should be refactored to utilize the time-series I/O staging library ADIOS2 [15].

ACKNOWLEDGMENTS

This work was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) under contract DE-SC0021299. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.* 22 (April 2010), 685–701. Issue 6. https://doi.org/10.1002/cpe.v22:6

[2] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. 2014. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 154–165. https://doi.org/10.1109/SC.2014.18

[3] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I. Ingólfsson, Joseph Koning, Tapasya Patki, Thomas R.W. Scogland,

Becky Springmeyer, and Michela Taufer. 2020. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems* 110 (2020), 202–213. https://doi.org/10.1016/j.future.2020.04.006

[4] AMD. 2022. ROCm System Management Interface. https://github.com/RadeonOpenCompute/rocm_smi_lib

[5] Inc. Apple. 2023. Activity Monitor User Guide. online. https://support.apple.com/guide/activity-monitor/welcome/mac

[6] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. 2013. There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 41, 12 pages. https://doi.org/10.1145/2503210.2503247

[7] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: performance introspection for HPC software stacks. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 550–560.

[8] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 47, 11 pages.

[9] David Boehme, Kevin Huck, Jonathan Madsen, and Josef Weidendorfer. 2019. The Case for a Common Instrumentation Interface for HPC Codes. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. 33–39. https://doi.org/10.1109/ProTools49597.2019.00010

[10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE (Ed.). Inria.fr, Pisa, Italy. https://doi.org/10.1109/PDP.2010.67

[11] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 2009. 24/7 Characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/CLUSTR.2009.5289150

[12] Intel Corporation. 2023. Data Parallel C++: the oneAPI Implementation of SYCL*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html#gs.3mmagp

[13] Todd Evans, William L. Barth, James C. Browne, Robert L. DeLeon, Thomas R. Furlani, Steven M. Gallo, Matthew D. Jones, and Abani K. Patra. 2014. Comprehensive Resource Use Monitoring for HPC Systems with TACC Stats. In *2014 First International Workshop on HPC User Support Tools*. 13–21. https://doi.org/10.1109/HUST.2014.7

[14] Hanhua Feng, Vishal Misra, and Dan Rubenstein. 2007. PBS: A Unified Priority-Based Scheduler. *SIGMETRICS Perform. Eval. Rev.* 35, 1 (jun 2007), 203–214. https://doi.org/10.1145/1269899.1254906

[15] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. ADIOS 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[16] Robert Hager, E.S. Yoon, S. Ku, E.F. D'Azevedo, P.H. Worley, and C.S. Chang. 2016. A fully non-linear multi-species Fokker–Planck–Landau collision operator for simulation of fusion plasma. *J. Comput. Phys.* 315 (2016), 644–660. https://doi.org/10.1016/j.jcp.2016.03.064

[17] Alex Harper. 2023. MenuMeters. online. https://ragingmenace.com/software/menumeters/

[18] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, Simone Chiesa, Bryan K Clark, Raymond C Clay, Kris T Delaney, Mark Dewing, Kenneth P Esler, Hongxia Hao, Olle Heinonen, Paul R C Kent, Jaron T Krogel, Ilkka Kylänpää, Ying Wai Li, M Graham Lopez, Ye Luo, Fionn D Malone, Richard M Martin, Amrita Mathuriya, Jeremy McMinis, Cody A Melton, Lubos Mitas, Miguel A Morales, Eric Neuscamman, William D Parker, Sergio D Pineda Flores, Nichols A Romero, Brenda M Rubenstein, Jacqueline A R Shea, Hyeondeok Shin, Luke Shulenburger, Andreas F Tillack, Joshua P Townsend, Norm M Tubman, Brett Van Der Goetz, Jordan E Vincent, D ChangMo Yang, Yubo Yang, Shuai Zhang, and Luning Zhao. 2018. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter* 30, 19 (apr 2018), 195901. https://doi.org/10.1088/1361-648X/aab9c3

[19] Dalibor Klusáček, Václav Chlumský, and Hana Rudová. 2015. Planning and Optimization in TORQUE Resource Manager. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 203–206. https://doi.org/10.1145/2749246.2749266

[20] Andreas Knüpfer, Christian Rössel, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E Nagel, et al. 2012. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*. Springer, 79–91.

[21] Linux man pages. 2023. htop(1) — Linux manual page. online. https://man7.org/linux/man-pages/man1/htop.1.html

[22] Linux man pages. 2023. proc - process information, system information, and sysctl pseudo-filesystem. online. https://man7.org/linux/man-pages/man5/proc.5.html

[23] Linux man pages. 2023. top(1) — Linux manual page. online. https://man7.org/linux/man-pages/man1/top.1.html

[24] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840. https://doi.org/10.1016/j.parco.2004.04.001

[25] Microsoft. 2023. Using Resource Monitor to Troubleshoot Windows Performance Issues. online. https://techcommunity.microsoft.com/t5/ask-the-performance-team/using-resource-monitor-to-troubleshoot-windows-performance/ba-p/375008

[26] Servesh Muralidharan. 2023. An overview of Argonne's Aurora Exascale Supercomputer and its Programming Models. online. https://extremecomputingtraining.anl.gov/wp-content/uploads/sites/96/2022/11/ATPESC-2022-Track-1-Talk-9-Muralidharan-Aurora.pdf

[27] NERSC. 2023. Perlmutter Architecture. online. https://docs.nersc.gov/systems/perlmutter/architecture/

[28] T. Newhouse and J. Pasquale. 2006. ALPS: An Application-Level Proportional-Share Scheduler. In *2006 15th IEEE International Conference on High Performance Distributed Computing*. IEEE, Paris, France, 279–290. https://doi.org/10.1109/HPDC.2006.1652159

[29] NVIDIA. 2020. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[30] OLCF. 2023. Frontier User Guide. online. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

[31] OLCF. 2023. Summit User Guide. online. https://docs.olcf.ornl.gov/systems/summit_user_guide.html

[32] OpenMP. 2022. OpenMP Specifications. https://www.openmp.org/specifications/

[33] Josko Plazonic, Jonathan Halverson, and Troy Comi. 2023. Jobstats: A Slurm-Compatible Job Monitoring Platform for CPU and GPU Clusters. In *Practice and Experience in Advanced Research Computing* (Portland, OR, USA) *(PEARC '23)*. Association for Computing Machinery, New York, NY, USA, 102–108. https://doi.org/10.1145/3569951.3604396

[34] Allan Porterfield, Rob Fowler, Anirban Mandal, David O'Brien, Stephen Olivier, and Michael Spiegel. 2012. *Adaptive scheduling using performance introspection*. Technical Report. TR-12-02. RENCI, 2012. https://renci.org/wp-content/uploads/2012/07/TR-12-02.pdf

[35] The Open MPI Project. 2023. Portable Hardware Locality (hwloc). online. https://www.open-mpi.org/projects/hwloc/

[36] Puppet, Inc. a Perforce Company. 2023. Welcome to Puppet Enterprise 2023.2. online. https://www.puppet.com/docs/pe/2023.2/pe_user_guide.html

[37] QMCPACK. 2023. QMCPACK miniapp: a simplified real space QMC code for algorithm development, performance portability testing, and computer science experiments. online. https://github.com/QMCPACK/miniqmc/tree/OMP_offload

[38] Robert B Ross, George Amvrosiadis, Philip Carns, Charles D Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K Gutierrez, Robert Latham, et al. 2020. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology* 35 (2020), 121–144.

[39] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 512–526. https://doi.org/10.1109/TPDS.2017.2766062

[40] S. Shende and A. D. Malony. Summer 2006. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications* 20, 2 (Summer 2006), 287–331.

[41] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[42] ORNL Tom Papatheodore. 2023. Hello jsrun. online. https://code.ornl.gov/t4p/Hello_jsrun

[43] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.