Efficient SDS Simulations on Multi-GPU Nodes of XSEDE High-end Clusters

Samuel Schlachter, Stephen Herbein, and Michela Taufer
Computer and Information Sciences
University of Delaware
Newark, DE 19716
Email: saschlac, sherbein, taufer@udel.edu

Shuching Ou and Sandeep Patel
Chemistry and Biochemistry
University of Delaware
Newark, DE 19716
Email: ousc, sapatel@udel.edu

Jeremy S. Logan
Remote Data Analysis and Visualization Center
Oak Ridge National Lab
Oak Ridge, TN 37996
Email: loganjs@ornl.gov

Abstract—Efficiently studying Sodium Dodecyl Sulfate (SDS) molecules’ formations in the presence of different molar concentrations on high-end GPU clusters whose nodes share accelerators exposes us to several challenges, including the need to dynamically adapt the job lengths. Neither virtualization nor lightweight OS solutions can easily support generality, portability, and maintainability in concert. Our solution complements rather than rewrites existing workflow and resource managers with a companion module that complements functions of the workflow manager and a wrapper module that extends functions of the resource managers. Results on the Keeneland cluster show how, by using our modules, accelerated SDS simulations more efficiently use the cluster’s GPUs while leading to relevant scientific observations.

I. INTRODUCTION

In this paper we address the problem of efficiently using high-end GPU clusters in which each node hosts multiple accelerators for fully atomistically resolved molecular studies. Simulations that apply fully atomistically resolved molecular models are more accurate than coarse-grained simulations and, therefore, are preferred for the study of thermodynamic properties in complex molecular systems despite being more computational demanding. More specifically, we use fully atomistically resolved molecular simulations to study the formation of Sodium dodecyl sulfate (SDS) molecules in the presence of different types of molar concentrations [1]. To study the SDS thermodynamics properties, we generate large numbers of long trajectories (jobs). Due to the fine-grained parallelism embedded in the molecular dynamics (MD) algorithms, the generation of long SDS trajectories benefits from accelerators (i.e., GPUs, FPGAs, Xeon Phi). At the same time, simulations on high-end clusters on which nodes share accelerators expose us to new challenges. For instance, these platforms do not allow the direct submission of jobs to an accelerator or the direct control of the job’s progression yet; instead, we have to rely on a CPU for these tasks. In the instance where nodes have multiple GPUs, one logical GPU job does not necessarily map to one physical GPU job. As a result, we must manually group multiple logical jobs into a single CPU job, and the CPU must distribute the jobs to the GPUs. This manual approach can lead to poor utilization of resources.

An efficient use of accelerators can be even more challenging for SDS simulations because their jobs can have dynamically variable runtimes based on the stage of the simulation and the molar concentrations. Figure 1 shows how the performance values of four SDS systems on Keeneland, measured in nanoseconds per day, differ among themselves and as the simulations evolve. Time constraints have to be considered when these simulations are performed on shared platforms such as an XSEDE cluster, e.g., jobs must have a maximum length of 24 hours. If job runtimes change as simulations evolve, the associated jobs can either be shorter than the maximum allowed length resulting in a loss of computing time or longer resulting in a premature termination of the simulations and the loss of results. More specifically, when several SDS jobs at similar simulation stages and with the same concentrations are manually assigned to the GPUs on the same node, they have similar execution times but their lengths, which may initially fit in the 24-hour time interval, can eventually grow to overcome the hard deadline of 24 hours. In this case, computation results between the last checkpoint and the deadline are lost (Figure 2.a). If randomly assigned to nodes, which may happen with a context unaware scheduler, SDS jobs with different concentrations or at different simulation stages can be assigned to the GPUs on the same node. In this case, some GPUs may complete their computation and have to wait for the other jobs to complete. Other GPU jobs may overcome the time limits and be terminated prematurely.

![Figure 1](image_url)
without storing the results (Figure 2.b). In addition, sudden failures can terminate one job leaving its GPU idle while the other GPUs on the same node are completing their work (Figure 2.c). All of these aforementioned conditions make the simultaneous pursuit of high accelerator utilization for fully atomistically resolved molecular trajectories of SDS systems particularly challenging.

![Diagram](image)

Fig. 2. Example of inefficiencies in SDS simulations when (a) jobs with the same concentrations are assigned to the same node but have constant initial performance; (b) jobs with different concentrations and constant initial performance are assigned to the same node; and (c) one job experiences early termination, leaving its GPU idle. Note that X = GPU in use but results not counted toward simulation progression; dash line = checkpoints; and red lines = GPU idle times.

Although some solutions based on virtualization or lightweight OS layers may help group diverse GPU jobs that are submitted in a bundle, these efforts are still in their infancy stage. When virtualization or lightweight OS layers are not available, resource managers cannot always isolate our GPUs from those of other users. This issue can arise either when we request one GPU but end up sharing the accelerator with other users or when we request one node with three GPUs and end up using only one GPU, thus preventing other users from accessing the idle accelerators. Resource managers also do not allow the simulations to easily propagate GPU errors back to the scientists. For example, in a resource manager such as TORQUE, errors relate to the CPUs that launch the GPU jobs rather than the GPUs that cause them.

To address the challenge, rather than rewriting existing workflow and resource managers, we propose to complement them with two modules, a Companion Module (CM) that complements functions of the workflow manager and a Wrapper Module (WM) that supplements functions of the resource managers. Our modules share the responsibility of monitoring jobs, predicting variable performance as simulations evolve, and assuring a higher utilization of the shared accelerators. As a proof of concept, we model the utilization of the SDS simulations with and without our modules; we observe significant improvements in utilization when our approach is in place. Ultimately, because of our modules, the SDS simulations more efficiently lead to results that are similar to those observed in wet labs.

The rest of this paper is organized as follows: Section II discusses important related work; Section III gives the overview of the two modules and how they predict variable performance; Section IV presents our simulations, estimates the maximal utilization with and without our modules; and outlines relevant scientific results; and Section V concludes the paper.

II. RELATED WORK

Currently available solutions for our simulations may or may not rely on virtualization. When clusters do include virtualization, such as when using Shadowfax [2], scientists can schedule isolated CPU/GPU pairs and associate failures with GPUs. However, virtualization imposes significant overhead in terms of power, performance, and noise or jitter. Moreover, when available, solutions based on virtualization are GPU-language specific, e.g., for CUDA only, and require the cluster to have a hypervisor and Virtual Machines (VMs). Therefore, on most high-end clusters, virtualization is not available. Alternative solutions to virtualization include lightweight, user-level implementations on Linux operating systems [3], [4], but these solutions are often dependent on a specific version of the GPU programming language or GPU generation. When virtualization or lightweight OS layers are not available, a workflow manager (e.g., Pegasus [5]) has to couple with a cluster resource manager (e.g., Torque) to run the simulations. Often, jobs have to be packed into customized bundles (one bundle for each node including as many jobs as GPUs) before being submitted to the resource manager. Note that we cannot schedule and monitor a job on a GPU directly, nor can we guarantee exclusive GPU access and control. Thus, there is certainly space for new solutions that integrate simple designs to work with existing workflow and resource managers while preserving generality, portability, and maintainability.

In pursuing these goals our solution proposed in this paper moves away from virtualization solutions. While doing so, our approach still supports diverse programming languages and accelerators. It also takes over the tedious work of the scientists of customizing the job bundles based on e.g., the type of simulations or node resources. Although we describe
our solution for the Keeneland cluster at Oak Ridge National Lab (ORNL) and we assess it in terms of maximum utilization for an ensemble of SDS simulations, it is expected to work on other XSEDE clusters and for other scientific workflow than molecular simulations alone. Work in progress includes the study of how our approach enables the generation of fully and partially synchronized trajectories on hybrid clusters for a broader range of simulations, e.g., for the study of carbon nanotube systems in different solvent concentrations [6].

III. FRAMEWORK

A. Modules’ overview

We propose to complement existing resource and workflow managers with two software modules that plug into those managers and, by interacting with each other, provide mechanisms to overcome the aforementioned constraints: a companion module and a wrapper module, both written in the Python programming language. The companion module runs on the head node of the cluster, accepts jobs from the workflow manager, instantiates children wrapper module instances, splits jobs into segments dynamically, and distributes the segments to the wrapper module instances. A wrapper module instance is launched on a computing node (one instance per back-end node of the cluster) as a resource manager job, receives and runs job segments from the companion module, and reports the status of job segments back to the companion module. The description of how the two modules interact with each other and with the resource and workflow managers follows in the next two sections.

B. Modules in action

Typically, when the maximum job length is limited to 24 hours, a workflow manager generates a set of 24-hour jobs either on the front-end of the cluster or on the user node while our companion module runs in an idle state on the front-end node of a cluster. This module intercepts the 24-hour jobs meant for the resource manager and, rather than forwarding the jobs to the resource manager, it generates one wrapper module instance per back-end node and submits those instances to the resource manager, as shown by the illustration in Figure 3.a. The resource manager launches the wrapper module instances as regular jobs on the back-end nodes (one per node). Each wrapper module instance starts on a back-end node and its lifespan is equal to the 24-hour time period imposed by the cluster administrators for each job. Once active, the instance requests job segments from the companion module directly; the instance can request as many job segments as GPUs on the node, as shown by the illustration in Figure 3.b.

The companion module collects important information on the state of the segments and the accelerators. It then uses this information to split the workflow manager’s jobs into smaller segments (here we use segments of up to 6 hours each) and sends a bundle of segments to each requesting wrapper module instance. Each bundle may contain as many segments as GPUs on its back-end node. As shown in the scenario illustrated in Figure 3.c, the requesting wrapper module instance runs the job segments on GPUs. When a segment is completed or encounters a sudden termination due to a system or application failure, the instance notifies the companion module by sending an information package that contains data on the execution status (e.g., success, system or application error), the segment run time, and the status of the accelerator on which the segment was executed. Partial results can also be integrated into this information package when needed. The companion module processes the information and uses the data to set up new segments. In the case of successful completion of a segment, a new 6-hour segment is sent. In the case of a failure, the companion module sends a new segment to the wrapper module with adjusted length to complete the initial 6-hour period and adjusts the checkpoint times accordingly, as shown in Figure 3.d where one of the segment fails and is replaced by a shorter one. If a segment overcomes the time period assigned to its execution, however, either because of a mis-prediction of the segment’s length or due to an increase of the segment’s length as the simulation evolves, the wrapper module reports the situation back to the companion module together with the state of the molecular system up to the last checkpoint. It is the task of the companion module to reschedule the part of the segment that was lost due to the premature termination, starting from the checkpointed state of the molecular system.

The communication between the companion and wrapper modules is repeated until either all the job segments are complete or the wrapper module reaches the end of its 24-hour lifespan. At the end of the 24 hours, if the companion module still has jobs waiting to be executed, this module submits an equal number of new wrapper module instances as the nodes needed to complete the jobs.

C. Estimating variable performance

Estimating the variable performance in nanoseconds per day of the SDS jobs is not trivial when the performance changes as the simulations evolve. In general, the way scientists decide how to set up the performance for a 24-hour job is as follows. The scientists run a reduced number of short tests on the cluster of interest and measure the performance. Based on these tests, they then hypothesize how the molecular system performs over time and start running it while keeping the performance value constant. Over time the performance may increase or decrease. In the first case, the jobs’ length may decrease resulting in idle accelerators; in the second case, the jobs’ length may increase resulting in premature terminations and the loss of results when the jobs exceed the 24-hour time limit. Too often, scientists realize the failure of their chosen performance only after multiple jobs prematurely terminate and the results are lost; in this case, they must select a new tentative optimal performance.

Our companion module uses heuristics to adjust performance in nanoseconds per day at runtime for the several segments. As with the standard approach described above, the module runs short tests on the cluster to measure the initial performance. It compares and matches the performance trend to a type of line (e.g., log, loglog, power, linear). With this information, it then builds an initial heuristic function \( f(t) \) and hypothesizes an initial performance. Contrary to the standard approach, at the end of every segment the wrapper module collects the performance data of that job and sends it to the companion module, which adds the new knowledge to a performance dataset and uses the collective information to adjust the heuristic function \( f(t) \). At the end of every
segment, the companion module uses this function to adjust the estimated performance as shown in Equation 1, where $t_{start}(i)$ is the start time of the segment and $t_{seg\text{length}}$ is the maximum length of the segment. The segment length is set up to last up to six hours and to cover part of the 24-hour time limit. This performance value in ns per day is then used to define the number of MD steps that will be simulated in the segment. Figure 4 shows the trend of our prediction heuristic with the estimated performance function $f(x)$ (red) when trying to follow the observed performance function $g(t)$ (blue) for the SDS system with a mole concentration of 1.0 on Keeneland. Similar trends were observed for the other concentrations. The figure also shows that our performance is able to follow the observed performance at runtime.

$$\text{performance}_{\text{estimate}}(i) = \int_{t_{\text{start}}(i)}^{t_{\text{seg\text{length}}}} f(t)\,dt \quad (1)$$

IV. SCIENTIFIC CASE STUDY

A. SDS molecular system

Traditionally SDS molecules have been studied in fields such as biochemistry, forensics, and biotechnology for their denaturing properties in unraveling proteins [7]. Recently SDS molecules have gaining new interest in the scientific community because studies indicate that SDS can play a key role in protein functions [8]. Here, we join this later effort...
by studying the SDS formation computationally. The computational study of SDS formations requires the generation of multiple trajectories in the presence of different types of molar concentrations. We consider four SDS systems and simulate their formations in multiple trajectories starting with random distributions of surfactant molecules. More specifically, the four systems have different initial concentration-dependent structures, each of which has a molar concentration of 0.1, 0.25, 0.5, and 1.0 respectively (Figures 5.a, .b, .c, and .d). For clarity, in these figures we removed the molecular waters that are treated explicitly in our simulations. Multiple trajectories are generated and analyzed for each of the four concentrations, each with a final length of 200ns and a temperature of 298K. Note that the simulations are currently in progress on Keeneland.

B. Simulation workflow and platforms

The study of the formation of SDS molecules generates a heterogeneous workflow across two different XSEDE clusters at ORNL, the GPU cluster Keeneland and the visualization and analysis cluster Nautilus. The workflow comprises two stages that are performed using different codes. The first stage, which is performed on Keeneland, benefits from GPUs because it relies on MD simulations and GPUs enable faster generation of longer binary trajectories. For this stage, we use the FEN ZI code [9], [10], which supports single-GPU MD simulations in NVT and NVE ensembles and energy minimization in explicit solvent. It also supports constraints on interatomic distances (e.g., shake and rattle, atomic restraints, and freezing fast degrees of motions) and uses accurate electrostatic interactions (i.e., Ewald summation). The second stage, which is performed on Nautilus, analyzes the trajectories to calculate the metrics of interest. For this stage, we use CHARMM [11] to process the FEN ZI trajectories. Because the focus of this paper is on the utilization and progression of trajectories on GPU platforms, we principally refer to the part of the study on Keeneland.

At the hardware level, each node on Keeneland has three NVIDIA M2090 GPUs. At the software level, TORQUE serves as a resource manager and Globus allows for the use of Pegasus as the workflow manager. The cluster uses a shared Lustre file system. The restrictions that users work with are as follows: a 24-hour time limit for each job, one job per node, and the non-isolation of GPUs. These conditions add several relevant constraints to the generation and management of SDS trajectories.

C. Modeling and assessing utilization

We use empirical observations from our SDS simulations on the Keeneland cluster to model the maximum utilization of the cluster’s GPUs with and without our modules. Note that by utilization we mean the time ratio accountable for a simulation’s progression. Most MD simulations include checkpoint mechanisms (often integrated into the application) that are commonly used to save the state of the molecular system every certain number of MD steps. If a job terminates without storing
the results to the proper files, e.g., because a sudden GPU failure occurs, the time between its last checkpoint and the termination is not counted toward the simulation’s progression. Moreover, if no mechanism such as the one we are proposing is in place, a single 24-hour bundle of jobs are submitted to a node. When one 24-hour job ends prematurely because, for example, a failure occurs, the impacted GPU remains idle until all the other jobs in the bundle complete.

Equation 2 presents the maximum utilization of a GPU cluster without our approach. In the equation $t_{\text{max}}$ is the maximum length of a job (24 hours on Keeneland). The time $t_{\text{arrival}}$ is equal to $t_{\text{lastchk}}$ if the job terminates successfully. Otherwise from the $t_{\text{max}}$ time, we subtract $t_{\text{arrival}} - t_{\text{lastchk}}$, which is the simulation time that we may lose if a failure occurs at $t_{\text{arrival}}$. In this case, the useful results are stored up to the last checkpoint time, $t_{\text{lastchk}}$. We also subtract $t_{\text{max}} - t_{\text{arrival}}$, which can be one of three possibilities: (1) the remaining time of the 24-hour period that we lose when the system or application fails at time $t_{\text{arrival}}$; (2) the time we lose when we underestimate the length of the job and $t_{\text{arrival}}$ is less than 24 hours; or (3) the time we lose when we overestimate the length of the job. Note that in the latter case, the job is truncated after 24 hours ($t_{\text{max}}$) and $t_{\text{arrival}}$ becomes $t_{\text{lastchk}}$. The arrival times ($t_{\text{arrival}}$) are estimated with the scientists’ approach, i.e., by using a reduced number of short tests and constant performance.

Equation 3 presents the maximum utilization of a GPU cluster when our modules are in place. The companion module splits each job into $n$ segments that are submitted to each GPU of one node over a 24-hour period. Each time a segment $i$ ends and a new segment $i + 1$ begins, the wrapper module instance has to pool the new computation from the companion module and restarts the simulation from the last checkpointed state of the molecular system (represented by $t_{\text{restart}}$). The time $t_{\text{arrival}}(i)$ is equal to $t_{\text{lastchk}}(i)$ if the segment terminates successfully; otherwise $t_{\text{arrival}}(i) - t_{\text{lastchk}}(i)$ represents the time lost due to a GPU failure. Although $t_{\text{max}} - t_{\text{arrival}}(i)$ in Equation 3 has the same meaning as $t_{\text{max}} - t_{\text{arrival}}$ in Equation 2, $t_{\text{max}} - t_{\text{arrival}}(n)$ refers to the $n$th segment of the bundle. We estimate the arrival times ($t_{\text{arrival}}(i)$) using Equation 4, where $G(t)$ is the integral of the observed performance function $g(t)$. Note that the function $g(x)$ was defined in the previous section and shown in Figure 4.

Failures of jobs may occur and have to be included in the utilization model. Premature terminations due to failures impact $t_{\text{arrival}}$ in Equation 2 and $t_{\text{arrival}}(i)$ in Equation 3. We observed that possible failures on Keeneland include: (1) file system failures such as IO interrupts, which refers to when an IO error interrupts a job that is in progress, and IO cascade, which refers to jobs that fail because their dependency had an IO error and the problem remained present; (2) wall-time failures when $t_{\text{arrival}}$ time is overestimated; (3) CUDA errors associated with unspecified launch failures and ECC errors; and (4) job failures associated with jobs aborted by the PBS server. In our sample of 830 SDS jobs, we observed a 10% rate of failing jobs. The majority of the failures (80% of the failing jobs) were associated to a Lustre file system outage that causes an IO interrupt job failure first and then a series of IO cascade errors until the outage is resolved. In addition, 11.1% of the failing jobs were associated to time mis-predictions, 7.4% to CUDA errors, and 1.2% to the PBS system. We use this data to model the time when failures occur over an interval of 24 hours; we fitted the empirical data from our SDS simulations on Keeneland to different distributions, and we observed that the more accurate distribution is the Weibull distribution for the observed failures. Figure 6 shows our fitting for the observed failures over 24 hours. The left y-axes show the occurrence of failures; the right y-axes show the probability density function (PDF); and the x-axes show the time in 24 hours. Given the data collected on Keeneland over three months (February - May 2013), we get the following probabilities:

- $p(\text{GPU job failure}) = 0.096$
- $\text{scale} = 0.0852843$
- $\text{shape} = 0.176396$

With this information, we can hypothesize the arrival of a failure and assign it to $t_{\text{arrival}}$ when our approach is not used in Equation 2 or to $t_{\text{arrival}}(i)$, with $i$ ranging from 1 to $n$ segments over the 24-hour interval when our approach is used in Equation 3. Empirically, we also observed that when a job does not fail, its length might vary with a very small deviation of less than 3% from its average length. Thus, we assume that each time a job or segment completes successfully, we have an associated 3% utilization loss.

Table I provides us with upper-bound estimates of the GPU utilization for twelve 10-day trajectories with four concentrations and three different seeds on Keeneland (three trajectories per node). Performances are selected with the more traditional approach followed by scientists (w/o our approach) and with our heuristic function (with our approach); the utilization is modeled using Equations 2 and 3 respectively. In Test I jobs with the same concentrations are manually assigned to the same node; in Test 2 jobs with different concentrations are randomly assigned to the same node. In both tests, failures occur based on the Weibull distribution. The table shows the
loss in utilization when different checkpoint frequencies are used (every 0.5, 1, 3, and 6 hours respectively) and when mechanisms such as the one we propose in this paper are not in place. As expected, the utilization decreases as the frequency of checkpointing grows. This phenomenon is more evident when our approach is not used because jobs are not split in segments with adjusted lengths and, thus, jobs can more easily overcome the 24-hour time limit or, when they fail, they leave their GPU idle for the rest of the 24 hours. In both tests, our modules provide us with a higher utilization of the GPUs, while assuring the automatic fitting of the jobs’ length at runtime. Intuitively, we can expect that larger idle times are associated with the lower utilization rates and that these idle times ultimately result in larger time gaps among the several trajectories as the simulations evolve. In other words, simulations are no longer progressing in concert, and while some trajectories are reaching orders of hundreds of nanoseconds, others may be several orders of magnitude behind.

### TABLE I. UPPER-BOUND UTILIZATION FOR TWELVE 10-DAY SDS TRAJECTORIES WITH FOUR CONCENTRATIONS AND DIFFERENT SEEDS ON KEENELAND WITH THE SCIENTISTS’ APPROACH (W/O OUR APPROACH) AND WITH OUR HEURISTIC FUNCTION (WITH OUR APPROACH). IN TEST 1 JOBS WITH THE SAME CONCENTRATIONS ARE ASSIGNED TO THE SAME NODE. IN BOTH TESTS FAILURES ARE PREDICTED BY USING A WEIBULL DISTRIBUTION.

<table>
<thead>
<tr>
<th>Checkpoint frequency</th>
<th>w/o our approach</th>
<th>with our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Test 1 (%)</td>
<td>Test 2 (%)</td>
</tr>
<tr>
<td></td>
<td>Test 1 (%)</td>
<td>Test 2 (%)</td>
</tr>
<tr>
<td>0.5 hr</td>
<td>94.36</td>
<td>87.99</td>
</tr>
<tr>
<td></td>
<td>99.53</td>
<td>97.23</td>
</tr>
<tr>
<td>1 hr</td>
<td>93.88</td>
<td>87.54</td>
</tr>
<tr>
<td></td>
<td>99.17</td>
<td>96.86</td>
</tr>
<tr>
<td>3 hr</td>
<td>91.97</td>
<td>86.10</td>
</tr>
<tr>
<td></td>
<td>97.85</td>
<td>95.25</td>
</tr>
<tr>
<td>6 hr</td>
<td>89.64</td>
<td>84.21</td>
</tr>
<tr>
<td></td>
<td>95.90</td>
<td>93.24</td>
</tr>
</tbody>
</table>

### V. CONCLUSIONS

The work presented in this paper is motivated by the fact that efficiently scheduling simulations on high-end GPU clusters still remain an open problem to be tackled. Virtualization is too costly to solve this problem, and lightweight user-level operating systems are in their infancy. To address this problem, our approach is designed to complement, rather than rewrite, existing workflow and resource managers. This solution to the problem utilizes a companion module that complements the workflow manager and a wrapper module that supports the resource managers. While doing so, our modules still support diverse programming languages and accelerators. We model the maximum utilization with and without our proposed approach for the simulations of the formation of SDS molecules. In light of our proposed solution, we observe that the utilization increases for the different SDS simulations, remaining above 93% even when heterogeneous-in-length jobs are assigned to the GPUs of the same node. For the same simulations we expect an utilization below 85%, when our modules are not in place. On-going simulations on Keeneland point out the formation of SDS molecules in agreement with experimental observations.

### ACKNOWLEDGMENT

This work was supported by the U.S. Army #ARO 54723-CS, NSF #941318, and by the NVIDIA University Professor Partnership Program.

### REFERENCES

Fig. 7. Four concentration-dependent structures with molar concentrations of (a) 0.1, (b) 0.25, (c) 0.5, and (d) 1.0 - after 22ns, 20ns, 20ns, and 15ns respectively.


