High Fidelity Data Collection and Transport Service Applied to the Cray XE6/XK6

Jim Brandt*, Tom Tucker†, Ann Gentile*, David Thompson§, Victor Kuhns‡, and Jason Repik‡

*Sandia National Laboratories, Scientific Computing Systems, Albuquerque, NM, USA
Email: {brandt|gentile}@sandia.gov
†Open Grid Computing, Austin, TX, USA
Email: tom@ogc.com
§Kitware Inc., Carrboro, NC, USA
Email: david.thompson@kitware.com
‡Cray Inc., Albuquerque, NM, USA
Email: {vgkuhns|jjrepik}@cray.com

***Abstract*—A common problem experienced by users of large scale High Performance Computer (HPC) systems, including the Cray XE6, is the inability to gain insight into their computational environments. Our Lightweight Distributed Metric Service (LDMS) is intended to be run as a continuous system service for providing low-overhead remote collection and on-node access to high-fidelity data, capable of handling 100s of data values per node per second, vastly exceeding the data collection sizes and rates typically handled by current HPC monitoring services while still maintaining much lower overhead. We present a case study of utilizing LDMS on the Cray XE6 platform, Cielo, to enable remote storage of system resource data for post run analysis and node-local access to data for run-time in-situ analysis and workload rebalancing. We also present information from deployment on an XK6 system at Sandia, where we leverage RDMA over the Gemini transport to further reduce LDMS overhead.**

I. Introduction

There is the desire within both the user and system administrator communities to have lightweight and scalable access to resource utilization related information at the node level that is out of band to the application. Such information can be useful from an application perspective for use in run-time reconfiguration of workload partitioning. From a user perspective it can be useful for detection of resource oversubscription and/or under utilization. It can also be used to drive future decisions in partitioning and resource allocation as well as provide deeper understanding of various performance degradation issues including some types of failure. System administrators can benefit from this type of information in both troubleshooting and assisting users in making more efficient use of platform resources. Finally, such information could benefit future system design making systems more robust and putting them in better alignment with user application needs.

Typical HPC monitoring systems, such as Ganglia [1] and Nagios [2] primarily target system administrator notification of failures and trend analysis and therefore are designed, in terms of overhead and infrastructure, for collection of a few tens of metrics per node with frequencies typically on the order of minutes to tens of minutes. In contrast, our Lightweight Distributed Metric Service (LDMS) software is intended to be compatible with both capacity and capability platforms and run as a system system service for providing low-overhead remote collection and on-node access to high-fidelity data, capable of handling 100s of data values per node per second. It targets collection not only of system monitoring data but also of application resource utilization data suitable for analysis and feedback.

In this paper, we present application of LDMS to the Cray XE6 and XK6 platforms, addressing configuration issues specific to these platforms and their specific features. We include a case study utilizing XE6 production HPC systems. This study was designed to show the viability of utilizing a system service (LDMS) to provide low-latency information directly to an application for run-time in-situ analysis and workload balancing decisions and to long term storage and analysis system for post-run analysis using the same Application Programmer Interface (API) for both types of operation.

LDMS supports both socket and Remote Direct Memory Access (RDMA) based transport. Socket based transport enables LDMS to be compatible with as many platforms as possible, however it comes at the cost of CPU overhead associated with per packet processing for both remote data requests and sends. This overhead can be eliminated by utilizing an RDMA based transport. While LDMS has RDMA support for Infiniband and 10Gb Ethernet networks the Gemini network presents some challenges in this area.

When the case-study work was performed LDMS had no RDMA over Gemini capability. Thus the case study results presented utilize LDMS using the socket transport on the Cray XE6 platforms "Cielo" located at Los Alamos National Laboratory and "Cielo Del Sur" (CDS) located at Sandia National Laboratories in New Mexico (SNL/NM).

LDMS has since undergone some architectural changes and now also supports RDMA over the Gemini network. This paper also presents some preliminary overhead information for the current LDMS architecture on an XK6 system, "Curie", at

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia, where we leverage RDMA over the Gemini transport to further reduce LDMS's CPU overhead on compute nodes.

We first present the architectural and topological configuration for the case study work performed on the Cielo and CDS systems. The remote storage and analysis for this work required the run-time transport of high-fidelity data being collected on Cielo to a Linux cluster located at Sandia National Laboratories in California (SNL/CA) and from CDS to four linux workstations also located at SNL/CA. We discuss how the LDMS architecture supports connectivities and transport across the asymmetric security domains encountered in this configuration. We then show how the same LDMS API also supports application access to node local system data. Finally we present overhead results with respect to CPU and memory for both the socket based transport utilized on the XE6 and the RDMA transport utilized on the XK6 system.

II. High Level LDMS Overview

LDMS is a data collection, transport, and storage tool that supports a wide variety of configurations. A high level diagram of the functional components used in the case study is shown in Figure 1. In the diagram data is collected on "Compute Nodes" though in general it may be collected on any device. LDMS uses a data pull model in general and the arrows in the diagram are showing direction of data flow. As shown in the figure there are two consumers of data being hosted by the LDMS daemons (*ldmsd*) running on the compute nodes: 1) application processes running on the compute nodes and 2) other *ldmsd*s running on "Aggregation Nodes". An in depth discussion of the applications' consumption of data is presented in Section III. The pulling of data by one or more *ldmsd*s from another can be repeated, asynchronously, in daisy chain fashion until the data reaches an endpoint shown in Figure 1 as "Storage Node(s)".

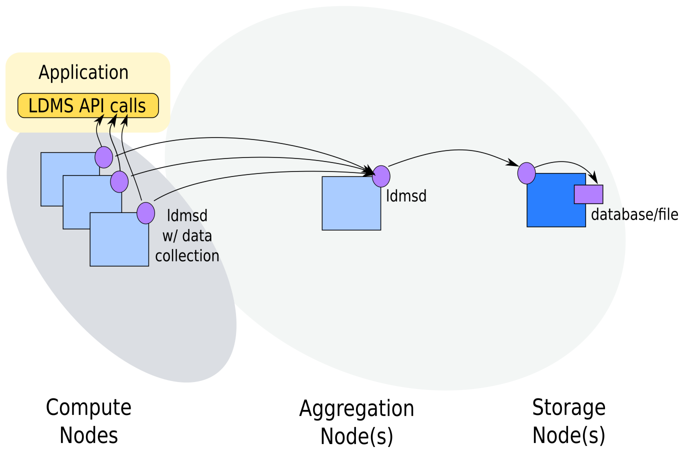


Fig. 1. LDMS data collection and transport system components. LDMS daemons collect and optionally aggregate, transport, and store data. Collected data can be accessed on-node by applications and transported off-node for aggregation and storage. On-node access and off-node transport utilize the same API.

In the LDMS framework, data are grouped into sets (called *metric sets*) of one or more values (called *metrics*) where each set is also associated with a "component identifier" (*component_id*). While these data can be collected/derived from any source and arbitrarily grouped, in this work such data and their names were primarily collected from the *proc* file system and were grouped according to their sources. Cray specific samplers include *kgnilnd* for data from `/proc/kgnilnd` and *gemctrs* which uses the Cray `gpcd` library (Gemini Performance Counter Device) to access the Gemini counter mmrs [3], [4] for Gemini tile and NIC data.

The example metric set below shows (`datatype, value, metricname`) tuples for data from `/proc/vmstat` collected from compute node "nid00574". *component_id* is a unique identifier for the component with which the data should be associated. While in this example *component_id* may seem redundant with the node name, a given node may have *metric sets* associated with node sub-components such as sockets, cores, communication buses, etc. In the case study work presented here each of these sub-components was identified with different ranges of *component_id*.

```
nid00574/vmstat
U64 0                component_id
U64 8205697          nr_free_pages
U64 1787             nr_inactive
U64 1939             nr_active
U64 657              nr_anon_pages
U64 822              nr_mapped
U64 3084             nr_file_pages
U64 0                nr_dirty
...
```

Note that each subsequent collection of a *metric set* overwrites the values from the previous collection and that no historic data is retained by an *ldmsd*.

In addition to the ability, mentioned above, to daisy chain *ldmsd*s, data can also be collected via any `ldmsd` along a chain. The data being hosted by any *ldmsd* can be accessed by any entity via the same API being used for the data pulls by daisy chained *ldmsd*s whether the accessing entity resides on the same node as the *ldmsd* being accessed or not. Relevant API calls in this work are `ldms_get_set`, `ldms_get_metric` and `ldms_get_u64` used to get a *metric set* handle, a *metric* handle from a set, and a value by type (`uint64_t` in this case) from that metric respectively.

III. Case Study: Resource-Aware Application Feedback Using LDMS

Performance of an application on a particular platform depends not only on the speed and capabilities of the hardware and system software but also on how the application utilizes those resources across all of its processes. The work presented in this section represents a project to assess the viability of enabling distributed HPC applications to utilize node level monitoring information to make run-time load balancing decisions.

For this case study, we present motivation, in-depth details of the applications used and their interaction with the LDMS monitoring system, platform and end-to-end configuration specifics, outcomes, and conclusions.

A. Motivation

Though it is the job of the operating system scheduler to place processes efficiently, the scheduler only has insight into processes running local to its node. Thus the burden of efficiently allocating work across all of the nodes associated with a particular application must fall to the application software and user. Further, for performance reasons, the typical practice in HPC is to bind processes to a particular core within a node leaving the job of balancing workload even within a node to the application/user. Thus tuning an application for performance requires some level of insight into how it will utilize the underlying compute resources both at the system level (e.g. nodes, network topology, storage) and at the node level (e.g. cpu, memory, cache, shared communication bus, network subsystem).

Due to the size and complexity of modern supercomputers it is difficult to gain insight into how these resources are being utilized by the application processes. Profiling tools such as OProfile [5], Tuning and Analysis Utilities (TAU) [6] and CrayPat [7], to name a few, allow a user to profile their applications but can incur significant overhead which can in turn impact the behavioral profile of the application. Also many of these tools require building instrumented code or relinking against other libraries. With complex codes the tools for automatically instrumenting an application may fail leaving the user to instrument by hand.

In order to gain low-level understanding of resource behavioral characteristics for the purpose of failure modeling, Sandia has been developing lightweight and scalable data collection, analysis, and visualization tools which target automated decision making based on relevant component level (node, core, network, etc.) information. Another project at Sandia has been targeting the use of node granularity resource utilization information to help guide applications in their selection of resource to load binding decisions.

The work presented in this paper combines tools from both of these projects to enable large-scale resource utilization information collection for post run analysis and run-time use of this information for guiding load balancing operations. The main goal of this work was to demonstrate the viability of large scale collection and use of this type of information to inform application load balancing decisions through both post-run analysis and run-time node/process local analysis.

B. Description

The LDMS data collection, off-node transport, and on-node interface provides the infrastructure to support resource-aware application feedback. An application can query LDMS during run-time for resource state data to be used in decisions, such as rebalancing and task stealing, either in response to an application's own dynamic requirements or in response to the demands of other applications competing for the same resources. Here we describe our experimental use of this enabling infrastructure applied to run-time repartitioning. We utilize production HPC applications and algorithms in this work, however determination of functional forms of values and conditions of interest and performance tuning was beyond the scope of this evaluation study.

The applications Aria [8] and Fuego [9], [10] in Sandia's SIERRA [11] multi-mechanics suite can, under certain conditions, repeatedly rebalance during run-time using the Zoltan [12] partitioner within the Trilinos Project's [13] algorithms for the solution of large-scale, complex multi-physics engineering and scientific problems. Zoltan determines partitioning taking into account specified "object weight(s)" and target relative "partition size(s)". Examples of the former include number of elements or particles. Aria and Fuego support some user selectable options for weight, imbalance thresholds to invoke rebalance, and rebalance frequency rates. In practice in these codes the target relative partition size is typically uniform; in this work we have modified Zoltan within SIERRA to call the node-local *ldmsd* to get system state information on demand and to use this information in setting the target relative partition size values at the core level. These function(s) for target partition size can be user defined. Thus a smaller (or larger) partition, with resulting lighter (or heavier) computational load, will be assigned to certain cores based on the their run time resource state information. In the Zoltan implementation used for this work, the entire space is repartitioned and finer-grained control based on, for example, system architecture, is not supported. While Zoltan includes a number of partitioning algorithms, RCB was used in all cases in this work. The Zoltan-LDMS interaction used to rebalance the applications of interest is illustrated in Figure 2.

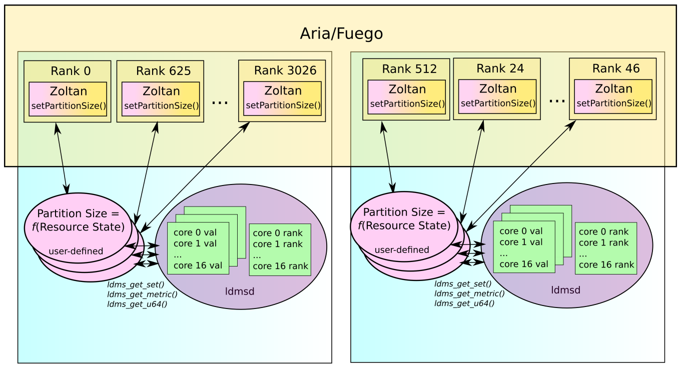


Fig. 2. Mechanics of the application feedback. Application uses Zoltan for repartitioning during run-time. Zoltan has been modified to use user-defined functions of the resource state data to calculate per-core target partition sizes to be used in the partitioning algorithm. Resource state data is obtained by LDMS data and is accessed on-node via the same API as is used to obtain data off-node for aggregation and transport.

Note that the LDMS update frequency is independent of the timescales of the Zoltan repartitionings. Zoltan determines target relative partition sizes based on the most current system state information held by LDMS. In this way there is minimal delay in obtaining the system state information, with the tradeoff of possibly using out-dated system information. Out-dated here refers to the time between when the last data sample was collected and when Zoltan queried its local *ldmsd* for the data. Thus the maximum age of the data obtained by Zoltan is bounded by the collection period. Note that our ultimate goal is not necessarily to achieve the *best* partitioning, but rather a *better* partitioning at minimal cost that doesn't offset the gains.

Determination of the best resource state information to be used in determining advantageous target relative partition sizes and the functional form for the computation is non-trivial; those utilized in this study were chosen naively in order to exercise the end-to-end functionality and to demonstrate the viability of this methodology. The expected mode of operation is that a more exhaustive post-run analysis of the resource data collected and stored off-node would be used to determine these.

C. Experimental Setup

Figures 3 and 4 depict a high level view of LDMS entities, their interaction with application processes, and the data paths between collection and endpoints (application and storage) used in this work on Los Alamos National Laboratory's Cielo and on Sandia National Laboratories' Cielo Del Sur (CDS). Both Cielo and CDS are Cray XE6 systems.

1) Platforms and LDMS Components: While the platform architectures and application path to monitored data are the same for both systems, the paths from compute nodes to storage is substantially different. These differences stem from both platform configuration and security posture and are discussed briefly here. Note that ideally LDMS would be run as a system service with aggregator *ldmsd*s being run on service nodes, distributed within the communication fabric so as to minimize network traffic hotspots. However, due to the experimental and transitory nature of our use of the systems for performing this work, we utilized hosts that enabled data transport to systems that were available for performing storage. In both cases we set out to store data to a distributed MySQL database. Since neither system was set up for this as a local option we had to write to remote storage devices in both cases and these had to be on systems with security postures compatible with the systems being monitored.

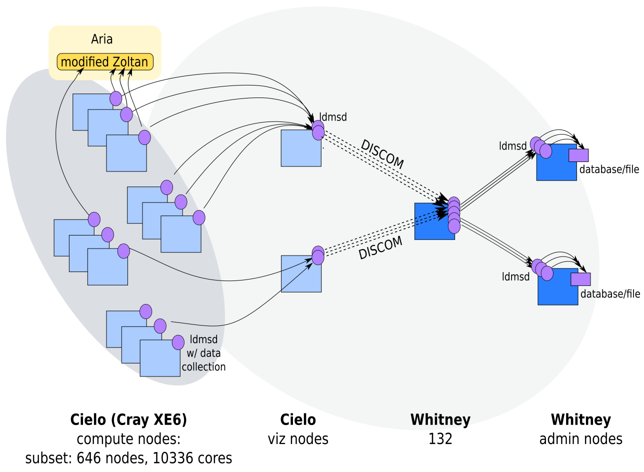


Fig. 3. LDMS data collection and transport system used for Cielo case study experiments.

On Cielo (Figure 3) we utilized visualization (viz) nodes for performing aggregation of compute node data due to their direct connectivity to compute nodes via the Gemini fabric and their external connectivity. Because the security posture of the viz nodes required that all external connections to viz nodes be initiated by the viz nodes themselves we had to enhance the LDMS transport to support this asymmetry. The need for this enhanced support stems from our use of a data *pull model*. Further details of this are presented in section III-C2 and depicted in Figure 5. The system that was available for storage of data from Cielo was Sandia's linux cluster Whitney. In this configuration we utilized the Whitney's gateway node, whitney132, to pass traffic from two of Cielo's viz nodes to two of Whitney's admin nodes which were used for storing data to SSD disks.

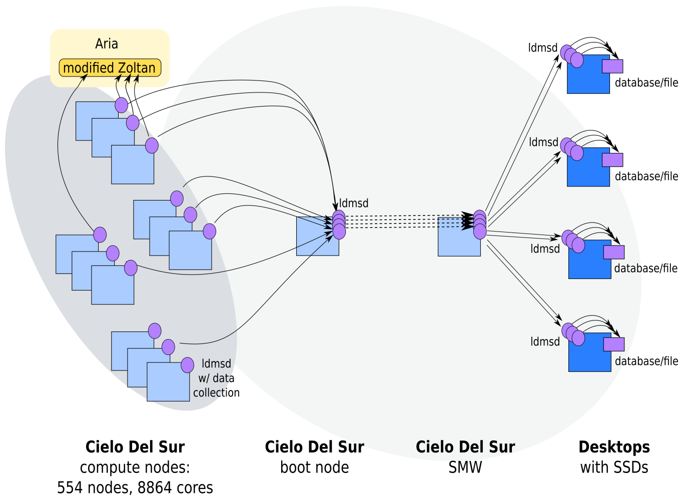


Fig. 4. LDMS data collection and transport system used for CDS case study experiments.

The CDS platform did not have any viz nodes and only had external connectivity to the login nodes and the System Management Workstation (SMW). The SMW, however did not have direct connectivity to the compute nodes. Thus we utilized the CDS boot node for running the *ldmsd*s that aggregated data from the compute nodes and the SMW for running *ldmsd*s to provide a transit path from the boot node to external storage nodes. In this case desktop "Shuttle" computers running Linux and using SSD disks for storage were used.

The difference in hosts being used for storage between Cielo and CDS was due to the difference in Cielo's and CDS's security postures and the need to have available and compatible storage hosts. In the case of Cielo the storage hosts we utilized from the Whitney cluster were a diskfull admin node and a diskless compute node. Each had four AMD Barcelona quad core 2.2 GHz processors and 32GB of memory. The admin and compute node were both running Red Hat Enterprise Linux 5 with a 2.6.18 kernel. We populated each of these two nodes with with two Crucial M4-CT256M4SSD2 SSDs for storing the data from Cielo. Due to the poor database performance of these machines we wrote the data initially to flat files and subsequently bulk loaded it into a database on the admin node for analysis. We collected 1012 data values per compute node at a sample interval of 9 seconds to exceed our self-imposed criteria of 6 data sets per minute per monitored node over at least 625 compute nodes.

For CDS our storage hosts consisted of four desktop Shuttle XPC PCs each with a single Intel Core i7 3.3 GHz processor and 6GB of memory and running Fedora 14 with a 2.6.35 kernel. We stored information to two MySQL databases per storage host. For database storage we used a single Crucial M4-CT256M4SSD2 solid state drive (SSD) per host. Thus our configuration used 8 databases (2 per SSD) across our four storage hosts. With this configuration we were able to ingest 1012 samples per node per 5 second interval over all 556 available compute nodes to distributed MySQL databases.

The XE6 platform, like most other production HPC platforms, only allows a user to schedule a single job on a node at a time. Thus in order to run our monitoring software concurrently with applications we ran it as a root process from the platform's boot node prior to running applications to be monitored.

2) Transport: LDMS supports both RDMA and socket based communication in Ethernet and Infiniband network environments. While the use of RDMA to transport data from compute nodes to the first level of aggregation minimizes the compute node CPU overhead incurred by *ldmsd*, the use case being presented used Cray's socket over Gemini transport for movement of data from compute nodes to aggregators because at the time the work was performed we did not have the capability to use RDMA over the Gemini network. LDMS now supports RDMA over the Gemini network and preliminary details are presented in Section IV-B. Transport of data from initial aggregators to our remote storage hosts also required use of the socket based transport because the target storage hosts were located at other sites multiple network hops away where the network was Gigabit Ethernet and did not support RDMA.

Support for asymmetric network access was added to LDMS to enable transport of data across arbitrary network topologies and specifically to enable use of the *pull* model where network security policy prevents connection setup in the desired direction. Figure 5 depicts the case study scenario described for the Cielo to storage host data path. There are three explicit modes of *ldmsd* operation: 1) Active, 2) Bridge, and 3) Passive. In the Active mode of operation the *ldmsd* listens for a connection request and engages the requesting entity in connection setup upon receiving the request (this is the normal mode of operation). In the event that the network security posture blocks external connection requests the *ldmsd* that would be listening must initiate the connection and hence is told what host it should be setting up a connection with; this is the Bridge mode of operation. In this mode the Bridge *ldmsd* periodically attempts to initiate a connection with a *ldmsd* running in Passive mode. The difference between a *ldmsd* operating in Passive mode and one operating in Active mode is that once the connection is established the Active *ldmsd* services requests for data over the connection in the case of a socket transport or does nothing if RDMA is being used. In the case of the Passive *ldmsd*, once the connection is established the Passive *ldmsd* performs queries for data from the Bridge *ldmsd* in the same manner that it would have done if pulling data from an Active *ldmsd*. It can be seen in the diagram of Figure 5 that independent of the mode of operation, which only affects the initial connection setup, data requests are made in the direction of the Compute Node while data flows in the direction of the Storage Host.

3) Core Specialization: Cray's "Core Specialization" [14] (*corespec*) mode enables a user to specify that a particular core on each compute node be utilized for all system related processes thus freeing the rest of the cores to be completely devoted to user application processes. The tradeoff is locality of application processes in that utilizing *corespec*, while eliminating system process competition with application processes, can increase the number of compute nodes required to be allocated in order for a user to procure a given number of

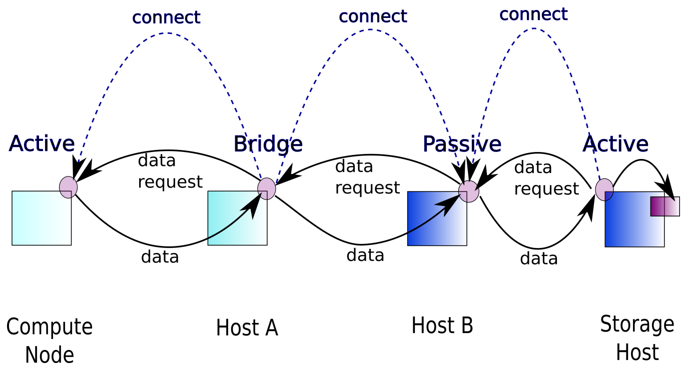


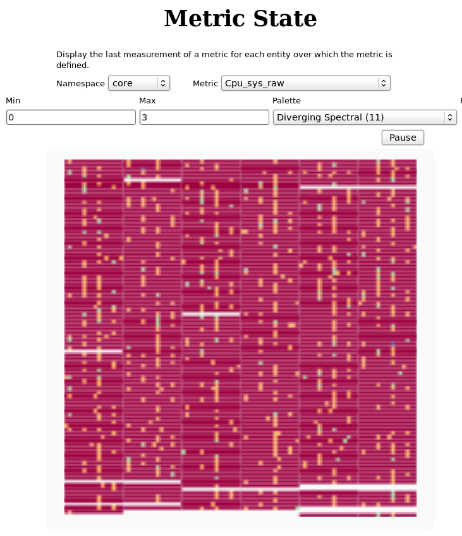
Fig. 5. LDMS supports asymmetric network access to satisfy security constrains on connection setup. While the overall data transport uses the pull model throughout, Active *ldmsd* listen for a connection request; Bridge *ldmsd* are used to set up a connection to Passive *ldmsd* which are prevented from initiating connections themselves.

cores and thus can also increase the application's remote communication. As part of this work we ran the Aria application in both normal and *corespec* modes to determine the effects on both system/application process contention and wall clock time to application completion. The application in this case utilized 8310 cores. Thus for the non-*corespec* case only 520 compute nodes were required while for the *corespec* case 554 compute nodes were required. We utilized the default of core 15 to host system processes when running in *corespec* mode.

Indeed the data collected to remote storage indicated that there were zero non-application processes and zero non-voluntary application process context switches on the 15 cores hosting application processes when running in *corespec* mode while distributions similar to those shown in Figures 6 and 7 were seen for non-application process system time and non-voluntary context switches respectively when not running *corespec*. Comparison of wall clock time to completion between the two cases for application runs of $\sim$ fifteen minutes duration showed minor improvement for *corespec* over non-*corespec* ($\sim 2.6\%$ in this case).

D. Demonstration

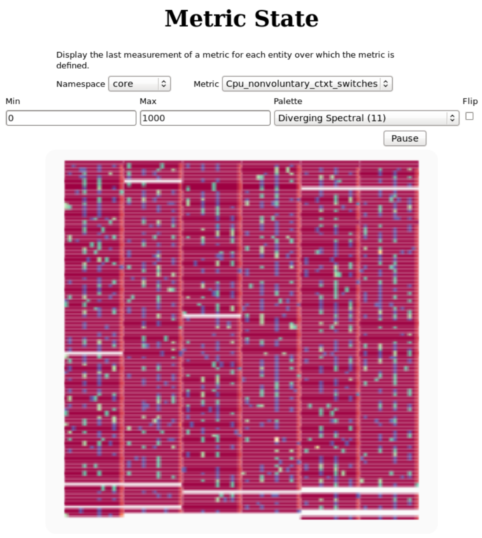
1) Small-scale Repartitioning: We first address the Fuego case. While the problem discussed here does not scale to significant sizes, it illustrates the dynamic application needs addressed by this work. Fuego's physics includes particle transport. Computational imbalance arises as articles move, are injected, split, and removed. Computational load varies in the physical space through time, and hence the workload per process will as well. Partitioning in the physical space for the problem spread across 32 core (32 chosen for visual clarity) is shown in Figure 8. At this time the partitions are unevenly divided in space, seeking to balance the particle density which is greatest at the bottom left of the figure. The regions of high particle number vary with time, as the particle locations and evolve inward and up and thus partition sizes and locations will change with time. As particles move in time the particle-partition mapping changes and the distribution is spread out thus triggering a rebalancing. In practice, Fuego is rebalanced using Zoltan when the particle imbalance across processes exceeds a user defined threshold.


Metric State

Display the last measurement of a metric for each entity over which the metric is defined.

Namespace: core
Metric: Cpu_sys_raw
Min: 0
Max: 3
Palette: Diverging Spectral (11)
Flip
Pause

Fig. 6. System time for one collection period (9 sec) on a per core basis over 556 nodes (8896 cores) of CDS. Each dot represents one core; each row of 16 dots represents all the cores of a node in numerical order Physical layout of the nodes is *not* representative of the rack layout. (Note that red is 0 and blue is 3)


Metric State

Display the last measurement of a metric for each entity over which the metric is defined.

Namespace: core
Metric: Cpu_nonvoluntary_ctxt_switches
Min: 0
Max: 1000
Palette: Diverging Spectral (11)
Flip
Pause

Fig. 7. Non-voluntary context switches for one collection period (9 sec) on a per core basis over 556 nodes (8896 cores) of CDS. Layout is that of Figure 6. (Note that red is 0 and blue is 1000)

In this work, we have augmented the determination of the target partition sizes to include a function of based on the ratio of idle cycles to total cycles utilized since the last partitioning. The intent is that processes that had exhibited larger idle time could take on more load (processes were bound to cores in all runs in this work). In practice we expect that a function that weights more heavily more recent load will be a better choice.

Figure 9 shows results for this example spread across 64 cores. In this simple demonstration there is a resultant small improvement in computational cycles dedicated to the application execution across all processes involved. Note that this example problem was generally well balanced to begin with (the y-axis does not start at zero). Detailed determination of data and the feedback functional form for this case were beyond the scope of this work and thus this should not be taken as a general indicator of potential results.

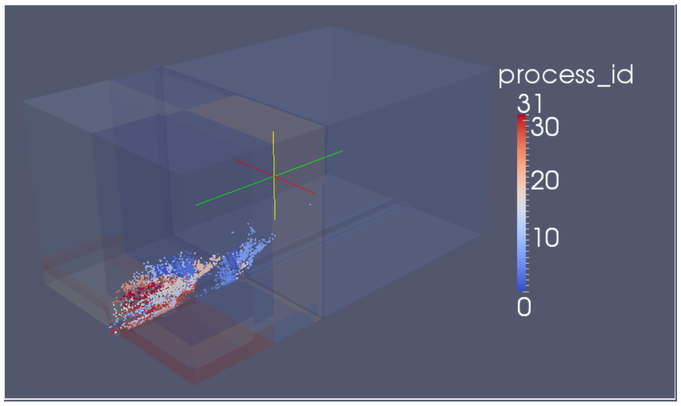


Fig. 8. Mapping of physical space and particles to partitions in a small-scale Fuego application. Repartitioning seeks to balance the computation. Regions of high particle number, and thus computational load, vary with time, starting at the lower left of the figure and evolving inward and up. Thus partition sizes and location will change with time. In this work, the usual repartitioning calculation based on particle number is augmented by consideration of run-time resource utilization data obtained by LDMS. The partitioner accesses LDMS data via the on-node API at run-time.

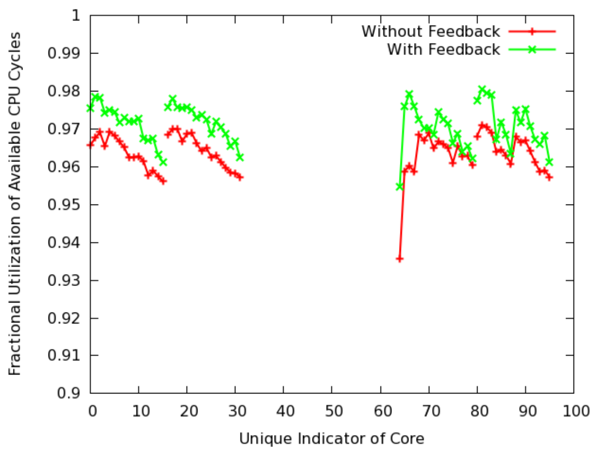


Fig. 9. Simple demonstration of shows improvement in computational cycles across all processes involved. Note that this example was generally well balanced to begin with (y-axis starts at 0.9).

2) Large-scale Repartitioning: Aria is a thermal code. In a general sense the dynamics of the simulation is not as closely tied to the computational load as in the particle case since, for instance, a hot spot in the calculation does not change the size of a matrix, but rather the values within it. However the problem may still be subject to imbalance and hence Aria provides the ability to rebalance using Zoltan when element imbalance or assembly time imbalance exceed user-defined thresholds.

Two cases were used for application feedback. We first performed a number of Aria runs and used LDMS for run-time collection of system utilization data and to transport the data off-node for subsequent analysis. This analysis indicated that non-voluntary context switches and interrupts occurred non-uniformly. Since these impact the time dedicated to the application on a resource, we chose to then assign target partition sizes based upon the number of occurrences of non-voluntary context switches and interrupts since the previous rebalancing. The revised application was then run on Cielo, with a maximum application size, and thus LDMS collection size, of 10112 cores.

The effect of using resource state data for run-time determination of the partition sizes is shown in Figure 10. In this case the distribution of partition sizes resulting was very broad resulting in a broad distribution of elements per partition as compared to the initial distribution. Non-voluntary context switches and interrupts occurred preferentially on Cores 0 and 9 during this run and as a result, the smaller partitions did occur preferentially on those cores. More work is required to determine a weighting of these quantities that might be advantageous for overall application run time.

We also applied the idle cycle criteria previously used in the Fuego problem to this problem on a 8310 core run on the Cray XE6 platform Cielo Del Sur at Sandia National Laboratories. We targeted a relatively comparable size, but still with less than the full complement of cores at our disposal, since this enabled us to also test the problem with Crays *corespec* option. Results of the partitioning with and without feedback are shown in Figure 11 for a set of selected timesteps (indicated in legend in parentheses). The non-feedback distribution changes only slightly in time, as expected. In the feedback case an early distribution (green) is much broader than the non-feedback distribution, probably to a larger degree than desired given the general computational balance of the problem. In this case the use of a feedback function that includes computational load drives the target partitioning and thus the distribution to a more balanced distribution.

E. LDMS Compute Node Overhead

The following data sources were utilized to collect base compute node related information: `/proc/meminfo`, `/proc/vmstat`, `/proc/stat`, `/proc/kgnilnd`, `/proc/interrupts`. Additionally, `/proc/PID/stat`, `/proc/PID/statm`, and `/proc/PID/status` were used to collect information about both application and LDMS process usage of compute node resources on a per-core basis.

In this study we collected 1012 data values per node, 720 of which were core specific, with a collection period of 9 seconds on Cielo, and a collection period of 5 seconds on CDS.

At the time this work was performed each data collector was run as a separate daemon process which then communicated with *ldmsd* via a unix domain socket. Thus CPU overhead was distributed across multiple cores of a node as determined by the Linux scheduler. Because the *ldmsd* processes were not bound to a specific core, except in the case of running *corespec* Section III-C3, they were seen to run on several cores throughout their run time. In order to determine the LDMS overhead all LDMS related processes were monitored and their user and system times, as presented from `/proc/PID/stat`, summed over the time window of an application run. Attribution of time to core mapping was achieved by summing the times a LDMS process was being run on a particular core as presented from `/proc/PID/status`

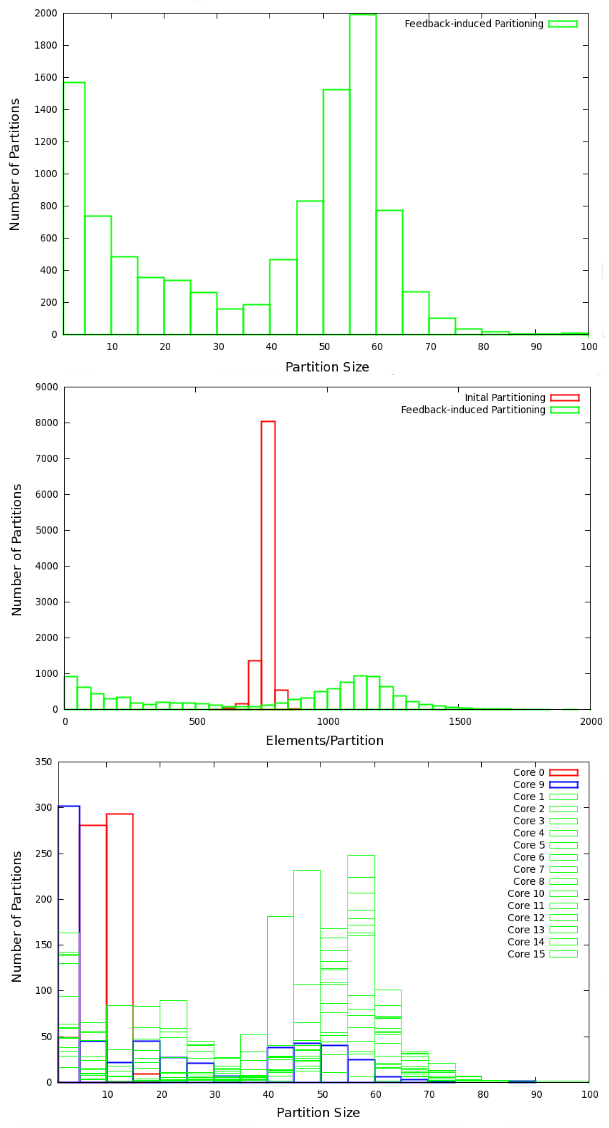


Fig. 10. Feedback Distributions for 10,112 core run of an Aria problem on Cielo at a selected partitioning step. Processes exhibiting more context switches and interrupts are assigned a smaller target relative partition size. In this case the distribution of partition sizes is broad (top) resulting in a broad distribution of elements per partition as compared to the initial distribution (middle). Non-voluntary context switches and interrupts occurred preferentially on Cores 0 and 9 across all nodes and, as a result, the smaller partitions occur preferentially on those cores (bottom).

over the same application run time window. Figure 12 shows the fractional utilization, by core, for all LDMS processes running on a particular node over an Aria application run. The average fractional utilization over all nodes involved in the Aria run, by core, is shown in Figure 13. Figure 14 shows the mean fractional CPU utilization over the same application run but includes high and low bars (note that for all cores the low is zero). Fractional utilization is just the LDMS process CPU time divided by the time window over which that time was summed (here the application run time).

Figure 15 shows the breakdown of CPU overhead by LDMS process (Note that this is presented as a % of node CPU time). It can be seen from this that the process with

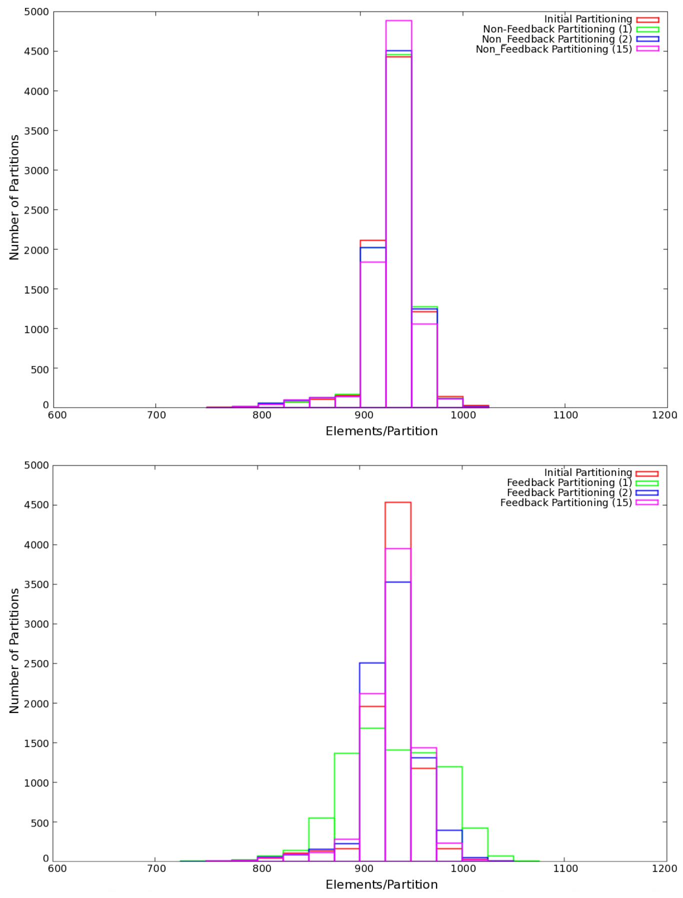


Fig. 11. Partition Distributions for selected timesteps for an 8310 core run of an Aria problem on CDS. Processes exhibiting larger ratio of idle cycles to total cycles utilized since the last partitioning are assigned a larger target relative partition size. Uniform partition sizes (top) results in tighter distributions than those without feedback. In the feedback case, when too broad partitioning occurs early (bottom, green), the partitioning feedback criteria self-corrects the distribution. Numbers in Legend in parentheses indicate timestep.

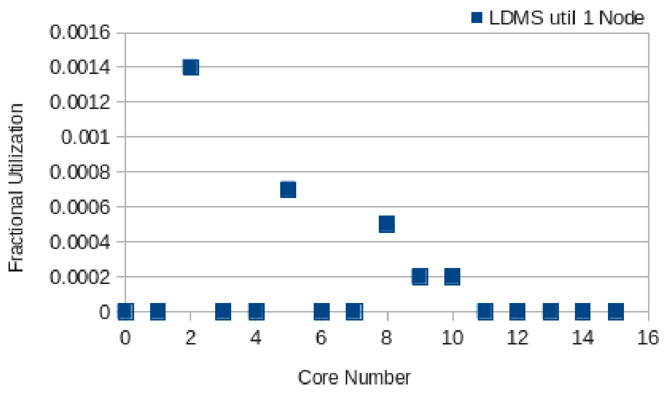


Fig. 12. Fractional utilization for LDMS processes on a per core basis running on a particular compute node in a 10,112 processor Aria run on Cielo.

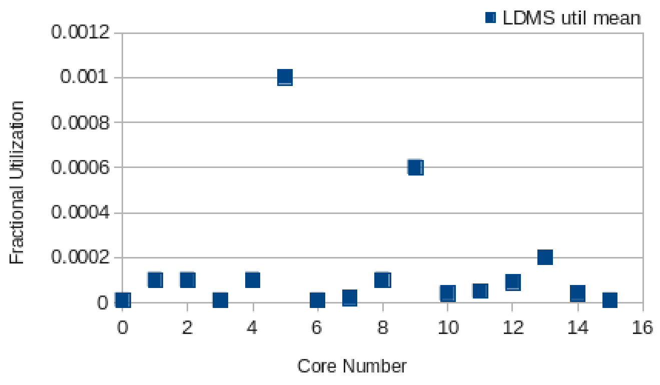


Fig. 13. Fractional utilization for LDMS processes on a per core basis averaged over all nodes involved in a 10,112 core Aria run on Cielo.

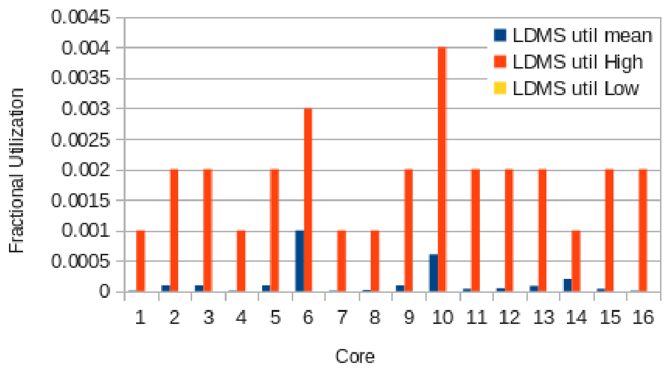


Fig. 14. Fractional utilization for LDMS processes on a per core basis, including high and low (0), over all nodes involved in a 10,112 core Aria run on Cielo.

the most overhead is *ldmsd* which not only gathers data from the collection daemons but responds to external queries for data updates from both application and aggregator *ldmsd*s. In this case the external *ldmsd* queries account for nearly all queries as the application queries are for several metrics on minute intervals while *ldmsd* aggregator queries are for all 1012 metrics every 5 seconds. The total over all LDMS processes was 0.009% of the compute node CPU time.

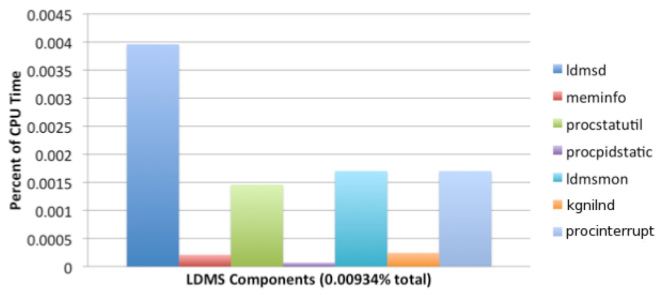


Fig. 15. Breakdown of CPU overhead by LDMS process presented as percentage of the node's CPU time on CDS. Total over all LDMS processes is 0.009% of the compute node CPU time.

Figure 16 shows the breakdown of memory footprint by LDMS process. Memory footprint here is defined as the "resident set size" taken from `/proc/PID/statm/` of each LDMS collector daemon process. Again *ldmsd* can be seen to have the largest footprint as it hosts all data sets. The aggregate memory footprint is the memory overhead incurred for all LDMS processes and is 4.02MiB. Given that the per-node memory size is 32GB this represents about 0.012% of the host memory.

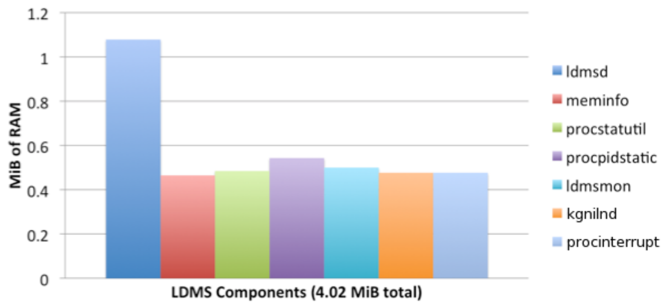


Fig. 16. Breakdown of Memory footprint by LDMS process on CDS. Aggregate memory footprint is $4.02MiB$ which is 0.012% of the host node memory size of 32GB.

The core LDMS infrastructure was re-written subsequent to the performance of this work and has several enhancements that reduce both overhead and complexity. Among these enhancements *ldmsd* now incorporates a plugin architecture which dispenses with the disparate collection daemons referenced above and incorporates them into the *ldmsd* process. Further details of this and other enhancements are presented in Section IV.

F. Conclusions

This case study demonstrated the viability of enabling distributed HPC applications to utilize node level monitoring information to enable run-time application feedback. The LDMS aggregation and transport capabilities enabled post-processing analysis of application resource utilization data and use of the on-node data interface in order to enable run-time repartitioning of Sandia HPC production applications. This work was performed on the Cray XE6 platforms Cielo and CDS at LANL and SNL in their production configurations. LDMS support for asymmetric network access enables transport of data across arbitrary network topologies, and includes features for connection setup that support the requirements of these systems security domains to transport data collected on the Cray XE6 platforms across DISCOM and the wide area to remote storage hosts.

While performance tuning was beyond the scope of this work, in separate work we continue to address the use of LDMS in conjunction with mapping and repartitioning algorithms. In particular we are addressing the inclusion of more architecture-specific details in the feedback and repartitioning algorithms.

Unlike typical monitoring systems, LDMS data is intended to collect data at frequencies suitable for providing meaningful application resource utilization information rather than those useful for general system monitoring or detecting failed nodes. Thus the target data collections rates of LDMS are of order seconds as opposed to those of, for example Gangia or Nagios, which are typically in the minutes to tens of minutes range. This requires that LDMS overhead must be very low. The compute node overhead on a per node basis for using the LDMS monitoring software in this case study was $\sim 0.01\%$ (on our non-Cray production systems Ganglia is over an order of magnitude higher than LDMS) with a memory footprint of $\sim 0.012\%$ of the host memory. In the next section we report on new features in LDMS, including those that take advantage of additional Cray-specific features to further reduce compute node overhead.

IV. LDMS Enhancements and Testing on the Cray XK6 platform

Subsequent to completion of the case study presented in this paper and as a result of some of the outcomes, parts of LDMS were re-designed and re-written to enhance LDMS with respect to complexity and overhead. These enhancements are briefly described in this section along with some preliminary overhead results from deployment on Sandia's two chassis Cray XK6 system, Curie.

A. Architectural Changes

Due to the complexity of running and managing multiple collection daemons in addition to the overhead of *ldmsd* maintaining socket connections to many daemon processes, *ldmsd* was modified to incorporate a plugin interface. This interface enables *ldmsd* to directly host plugins for performing data collection as well as for other purposes (e.g. storage operations). A Unix Domain Socket based interface enables instantiation and configuration of plugins which are written as separate libraries. Additionally several serial operations with respect to both collection and aggregation of data have been parallelized thus reducing the time window required for compute node *ldmsd*s and aggregator *ldmsd*s to update their information.

As a result of these changes the memory footprint has been reduced (e.g. *ldmsd* with 5 collector plugins now has a footprint of about 1.3MB as opposed to approximately 3.3MB for *ldmsd* and the same 5 collectors as daemons. Additionally the aggregate CPU overhead has also been reduced. Results are presented in Section IV-C below.

B. RDMA over the Gemini Network

Perhaps the most substantial enhancement to LDMS, with respect to compute node CPU overhead on Cray XE6/XK6 platforms, since the case study was performed, has been the implementation of an LDMS Gemini based RDMA transport. This transport was enabled by enhancements to the Cray Linux Environment that allow configuration of system "protection domain tags" (pTags) [15] for application use. The RDMA transport enables an aggregation *ldmsd* to directly fetch data from a compute node *ldmsd* without CPU intervention. This means that the only CPU overhead on the compute node, when using this transport, comes from data collection plugins. The benefit can be seen by comparing the CPU overhead of an *ldmsd* being queried by an aggregator *ldmsd* over socket vs. RDMA transports while collecting identical metrics at identical collection periods. Overhead results are presented in Section IV-C below.

The fan-in capability of *ldmsd* aggregation nodes is currently limited by the size of the Gemini NICs memory registration table to $\sim$ 900 *metric sets*. This limitation will be alleviated by the *ldmsd* RDMA transport registering larger memory regions and managing connections there as opposed to the current registration per *metric set* scenario.

C. Gemini RDMA Overhead

Figure 17 shows the number of metrics for each of the five samplers run in the experiments on Curie. Of these, four read from the `proc` filesystem (*meminfo, vmstat, procstatutil, kgnilnd*) and one (*gemctrs*) makes an `ioctl` call to get its data.

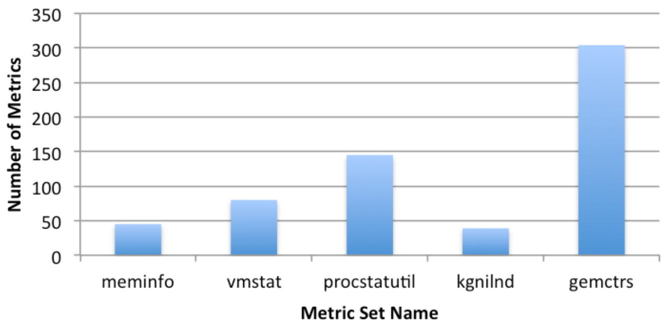


Fig. 17. The number of metrics for each named metric set.

By referring to both Figures 17 and 18 it can be seen that CPU overhead scales up as the number of metrics in a *metric set*. The *procstatutil* sampler is seen to have greater overhead than would be expected given the number of metrics being collected and bears further investigation. The difference in overhead with respect to the *gemctrs* sampler is likely due to the difference in the data source and method of acquisition of this data (i.e. `ioctl` call vs. read from `proc`).

Figure 18 provides a comparison of overhead associated with samplers and transports. From this figure it can be seen that: 1) overhead is mostly due to the on-node data collection process, 2) the RDMA (UGNI) transport overhead is the same as that running with no data being pulled from the compute node *ldmsd* (there is a 2 to 3 jiffy startup cost associated with *ldmsd* using the UGNI transport), and 3) the RDMA transport overhead is less than that for the SOCK transport in all cases.

Figure 19 provides a comparison of the memory footprints associated with the same samplers and transports referred to above. It can be seen from this that in all cases that *ldmsd* using the RDMA transport has a larger memory footprint than using SOCK and that the samplers, with the exception of *gemctrs*, all contribute about the same to the increase over *ldmsd* running alone. It can also be seen by comparing this figure with Figure 16 that the redesigned *ldmsd* architecture has a substantially smaller memory footprint than the previous daemon based collector architecture.

V. Conclusions

Our Lightweight Distributed Metric Service (LDMS) software is intended to be run as a system service for providing low-overhead remote storage of and on-node access to high-fidelity system related data, suitable to enable post-processing analysis of application resource utilization data and use of the on-node data interface in order to enable run-time application feedback. It provides end-to-end capabilities for data collection, transport, and storage.

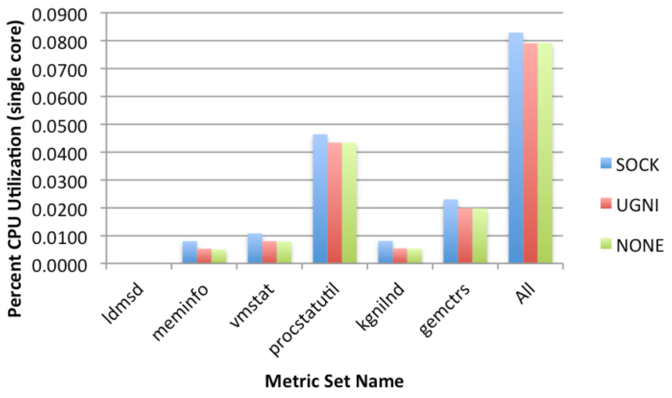


Fig. 18. Percent utilization of one core for ldmsd running each of 5 metric set samplers, all 5 simultaneously, and none. These cases are shown for *ldmsd*s being queried over the socket (SOCK) and RDMA (UGNI) transports on one second intervals as well as not being queried at all (NONE)

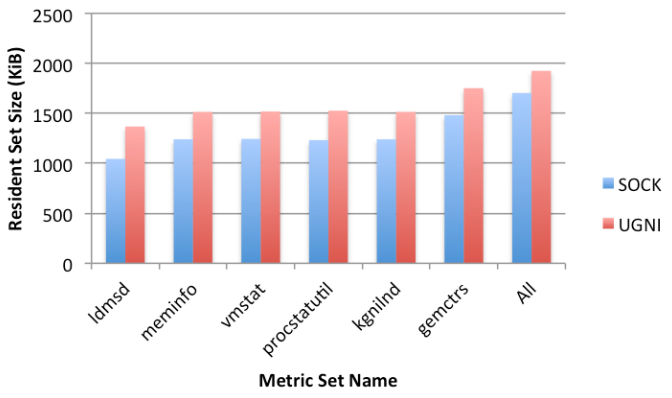


Fig. 19. The Resident Set Size (RSS) for *ldmsd* running each of 5 metric set samplers, all 5 simultaneously, and none. These cases are shown for *ldmsd*s being queried over the socket (SOCK) and RDMA (UGNI) transports on one second intervals.

In this work, we have utilized LDMS on production XE6 platforms in the demonstration of its viability for use in analysis and run-time repartitioning of production HPC applications. We have demonstrated features of the LDMS communication architecture to support the asymmetric network security domains in which production Cray systems exist at SNL and LANL.

We have presented the memory and CPU related overhead of LDMS and how impact on applications can be minimized in the Cray XE6/XK6 environment by taking advantage of Cray specific features such as *corespec* and RDMA over Gemini.

Acknowledgments

The authors would like to thank the following:

- Regarding platforms used in the case study: John Zepper, John Noe, Jeff Ogden, Sue Kelly and Bob

Ballance (SNL) for access to Cielo Del Sur; Jim Lujan, Cory Leuninghoener, Kathleen Kelley, Cindy Martin, Quellyn Snead (LANL), and Joel Stevenson (SNL) for assistance in the Cielo deployment; and Jerry Friesen, Adam Supinger, and Tuesday Armijo (SNL) for access to Whitney.

- Larry Kaplan and Jason Schildt (Cray) for XE6 platform specific information used in the case study.
- Regarding the applications utilized in the case study: Karen Devine (SNL) for information on Zoltan; Pat Notz, Greg Wagner, Stephan Domino, Alex Brown, and Robert Baca (SNL) for information on SIERRA codes; Jonathan Hu (SNL) for information on SIERRA-Trilinos integration.
- Ryan Olsen, Steve Martin (Cray), and Hasan Abbasi (ORNL) for information on UGNI.
- Kevin Pedretti (SNL) for information on the use of the `gpcd` library.

REFERENCES

[1] "Ganglia," http://ganglia.info.

[2] "Nagios," http:/nagios.org.

[3] Cray Inc., "Cray Linux Environment (CLE) 4.0 Software Release," Cray Doc S-2425-40, 2010.

[4] ——, "Using the Cray Gemini Hardware Counters," Cray Doc S-0025-10, 2010.

[5] OProfile, "OProfile," http://oprofile.sourceforge.net/news.

[6] U. of Oregon, "Tuning and Analysis Utilities: TAU," http://www.cs.uoregon.edu/Research/tau/home/php.

[7] Cray Inc., "Using Cray Performance Analysis Tools," Cray Doc S-2376-52, 2011.

[8] P. Notz, S. Subia, M. Hopkins, H. Moffat, and D. Noble, "Aria 1.5 User Manual," Sandia National Laboratories Report SAND2007-2734, 2007.

[9] S. P. Domino, C. D. Moen, S. P. Burns, and G. H. Evans, "SIERRA/Fuego A Multi-Mechanics Fire Environment Simulation Tool," in *41st AIAA Aerospace Sciences Meeting (AIAA Paper 2003-0149)*, 2003.

[10] S. Domino, G. Wagner, A. Luketa-Hanlin, A. Black, and J. Sutherland, "Verification for Multi-Mechanics Applications," in *48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (AIAA Paper 2007-1933)*, 2007.

[11] H. C. Edwards, "Sierra Framework Version 3: Core Services Theory and Design," 2002.

[12] K. Devine, E. Bowman, R. Heaphy, B. Henrickson, and C. Vaughan, "Zoltan Data Management Services for Parallel Dynamic Applications," in *Computing in Science and Engineering*, vol. 4, no. 2, 2002, pp. 90–97.

[13] Sandia National Laboratories, "The Trilinos Project," http://trilinos.sandia.gov.

[14] Cray Inc., "Workload Management and Application Placement for the Cray Linux Environment," Cray Doc S-2496-31, 2010.

[15] ——, "Using the GNI and DMAPP APIs," Cray Doc S-2446-4003, 2012.