Demonstrating Improved Application Performance Using Dynamic Monitoring and Task Mapping

J. Brandt, K. Devine, A. Gentile, and K. Pedretti
Sandia National Laboratories
Albuquerque, NM, 87185
Email: (brandt|kddevin|gentile|ktpedre)@sandia.gov

***Abstract*—This work demonstrates the integration of monitoring, analysis, and feedback to perform application-to-resource mapping that adapts to both static architecture features and dynamic resource state. In particular, we present a framework for mapping MPI tasks to compute resources based on run-time analysis of system-wide network data, architecture-specific routing algorithms, and application communication patterns. We address several challenges. Within each node, we collect local utilization data. We consolidate that information to form a global view of system performance, accounting for system-wide factors including competing applications. We provide an interface for applications to query the global information. Then we exploit the system information to change the mapping of tasks to nodes so that system bottlenecks are avoided. We demonstrate the benefit of this monitoring and feedback by remapping MPI tasks based on route-length, bandwidth, and credit-stalls metrics for a parallel sparse matrix-vector multiplication kernel. In the best case, remapping based on dynamic network information in a congested environment recovered 48.9% of the time lost to congestion, reducing matrix-vector multiplication time by 7.8%. Our experiments focus on the Cray XE/XK platform, but the integration concepts are generally applicable to any platform for which applicable metrics and route knowledge can be obtained.**

I. Introduction and Related Work

We demonstrate the benefit of combining dynamic system monitoring, architecture-specific network data, and task management algorithms to avoid application performance degredation caused by competing applications. Clearly, an application's performance depends strongly on its ability to use optimally the resources allocated to it. However, many factors influencing performance are out of the application's control. One such factor in a shared parallel computing environment is competing applications' use of shared network resources. When several applications communicate over the same network links, congestion can cause slow-downs in application performance. Performance degredation in such cases is well known anecdotally (e.g., every sysadmin has heard something like "My app runs more slowly whenever Joe is running jobs, too."), and also has been well documented [1].

In this work, we aim to relocate or remap MPI tasks among nodes to reduce the impact of network congestion on an application's performance. We assume an application is allocated a set of nodes by the system's resource manager. Our work does not influence the selection of these nodes, although other efforts have looked at strategies for reducing congestion by using static system topology information to more selectively allocate and order nodes (e.g., [2]–[4]). MPI provides a default mapping of ranks to cores in allocated nodes, typically a simple linear ordering of ranks, with no regard for the communication patterns of the application. Given a set of allocated nodes, our goal is to find a new assignment of MPI ranks to the nodes' cores that attempts to minimizes the cost of communication for the application. We call this assignment a "mapping" or "task placement."

In a non-shared environment, this mapping can examine distances or connections between nodes to assign interdependent application tasks to nodes that are "nearby" in terms of proximity or topology, decreasing the cost to communicate data between the tasks. For example, grouping interdependent tasks within cores of nodes as much as possible can significantly reduce communication costs in large-scale systems [5], and further accounting for proximity of nodes within the network topology can extend scalability to larger core counts [6]. Several software tools exist to compute such mappings, including graph-based mapping in LibTopoMap [7], Scotch [8], and Jostle [9], and geometric mapping MJ in Zoltan2 [6]. All rely on the application to provide some models of the network and application communication patterns to perform mapping; LibTopoMap, additionally, gets node-level information from the hwloc tool [10]. The parallel run-time environment Charm++ performs topology-aware mapping for grid and torus networks, approximating communication costs by "hop-bytes": the distance between nodes in the network multiplied by the number of bytes sent [11], [12]. In this work, we have chosen to use a graph-based mapping strategy in order to explicitly account for the cost of communicating within the network. We opted to use Scotch, but other packages could be used similarly without loss of generality.

Additional complexity arises in shared computing environments, where applications cannot account *a priori* for environmental factors such as competing network traffic. In such cases, dynamic information from the computing environment is needed. System-level software is needed to collect metrics of interest (e.g., bandwidth or stalls metrics), aggregate them into useful form, and deliver them to applications. Several sampling and aggregation tools exist that could provide the base data for this functionality; see, for example, MRNet [13], Ganglia [14], [15], Nagios [16], Performance Copilot [17], and the Lightweight Distributed Metric Service (LDMS) [18]. A performance comparison of these tools is beyond the scope of this paper, and, indeed, is orthogonal to this discussion, as any

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.

tool with sufficient data collection and aggregation capabilities could be used. We have chosen to use LDMS because of its excellent scalability [18] as demonstrated by its deployment on NCSA's 27648-node Cray XE/XK platform (Blue Waters) [19], as well as its existing deployments on our local Cray XE/XK systems. Moreover, LDMS is open source, which will give us greater flexibility in interacting with dynamic data in future work. Additional capability is also required for assessing the base data in the context of the system architecture and making it available to mapping tools.

Dynamic information can then be used in graph-mapping algorithms to set graph-edge weights representing communication costs in the network. Algorithms that attempt to minimize application communication costs then have a real-time picture of network performance, allowing more effective mapping decisions to be made.

While we focus on task placement in this work, the integration strategies described could also be applied to other resource management scenarios. For example, dynamic information and applications' expected communication patterns could both be integrated with existing static information in batch scheduling systems to reduce contention at a system level. Or dynamic computational load and network information could be used for data-level load balancing within applications, adjusting processor work loads to accommodate, say, power throttling or localized network congestion. Many opportunities exist for performance improvements once the mechanisms for collecting and sharing dynamic system information are in place.

This paper makes several contributions.

- We outline the key components needed to integrate dynamic monitoring with resource management tools.
- We describe a specific implementation of an integrated monitoring, analysis and response environment for dynamic task placement using LDMS and Scotch to avoid network congestion in a Cray XE/XK system.
- We demonstrate the potential benefits of this integrated system to reduce execution time of a key computational kernel – sparse matrix-vector multiplication.

We target Cray XE/XK systems since they are widely used for large-scale scientific computing; NERSC's Hopper and Los Alamos' Cielo are both Cray XE/XK systems. Many concepts from this paper would apply for other architectures given appropriate data collection tools for those systems.

This paper continues as follows. In Section II, we address issues in the Cray Gemini environment that motivate both the need for dynamic response to resource state and the need for system-level monitoring to acquire relevant data. In Section III, we outline the key components of an integrated monitoring and response system, and describe our specific implementation for resource- and architecture-aware task mapping. In Section IV, we describe our experimental set up for generating controlled resource-state variations and dynamic mapping response. Results of the experiments are given in Section V. We address future work and conclude in Section VI.

II. The Cray Gemini Network

All High Performance Compute (HPC) systems employ a variety of resources that are shared among nodes within a job and among all jobs running across the system. Examples include high speed network interconnects and file systems. Typically these resources are insufficiently provisioned to avoid some level of contention when there are many nodes concurrently competing for them.

Figure 1 illustrates the Cray Gemini [20], [21] interconnect utilized in their XE/XK platforms. (In particular, this figure represents the 64-node, 32-Gemini system, Curie, used in this work.) Typically this interconnect is configured as a 3D torus; the wrap-around links have been eliminated in the figure for clarity. In the figure, each circle represents a Gemini routing element, each of which connects directly to two host nodes. Gray circles denote Gemini associated with service nodes which are unavailable to applications, while blue indicates association with compute nodes. The XYZ mesh coordinates associated with Gemini are used as their labels. Two hosts associated with each Gemini are indicated by comma separated pairs above and to the left of each Gemini. Hosts have unique node identifiers called their nid numbers. This identifier has the same format for all hosts; it is the word `nid` followed by a five-digit host number padded with zeros to the left (for example, hosts in the front lower left of the figure denoted by 62,63 have identifiers `nid00062` and `nid00063` respectively). The gray bi-directional arrows depict the network connections between adjacent Gemini. The orange and red arrows illustrate traffic routes taken between a few of the Gemini.

Routing between any two Gemini is deterministic and goes first from source X to destination X, then from source Y to destination Y, and then from source Z to destination Z. The shorter of the paths in the +/- directions is taken. In case of a tie, the + direction is taken if the matching destination coordinate is even, and - if it is odd. As an example, traffic sent from `nid00012` to `nid00038` proceeds (as shown in the figure) by the following path: $(0,0,6)$ $X+ \rightarrow (1,0,6)$ $Y+ \rightarrow (1,1,6)$ $Z- \rightarrow (1,1,5)$ $Z- \rightarrow (1,1,4)$ $Z- \rightarrow (1,1,3)$.

To handle link congestion without data loss, the Gemini network utilizes a credit-based flow control scheme. A source is allowed to send only an amount of traffic to a particular destination for which it has credits. When a source runs out of credits for a destination but has data to send, it must pause (*stall*) until it receives credits back from the destination. Stalls when sending from one Gemini to another are referred to as *credit stalls*. Likewise, within the Gemini router, a credit-based flow control scheme is utilized to ensure that data in input buffers cannot be transferred to internal output buffers unless there is space available. Stalls in this case are called *inq stalls*. Thus, depending on which host resources have been allocated to which applications, there can be contention for network resources, potentially causing congestion, which can adversely affect application performance across multiple jobs. Additionally, within the Gemini distributed network fabric, different link types have different maximum bandwidths (not shown in the figure), further exacerbating link oversubscription and network congestion.

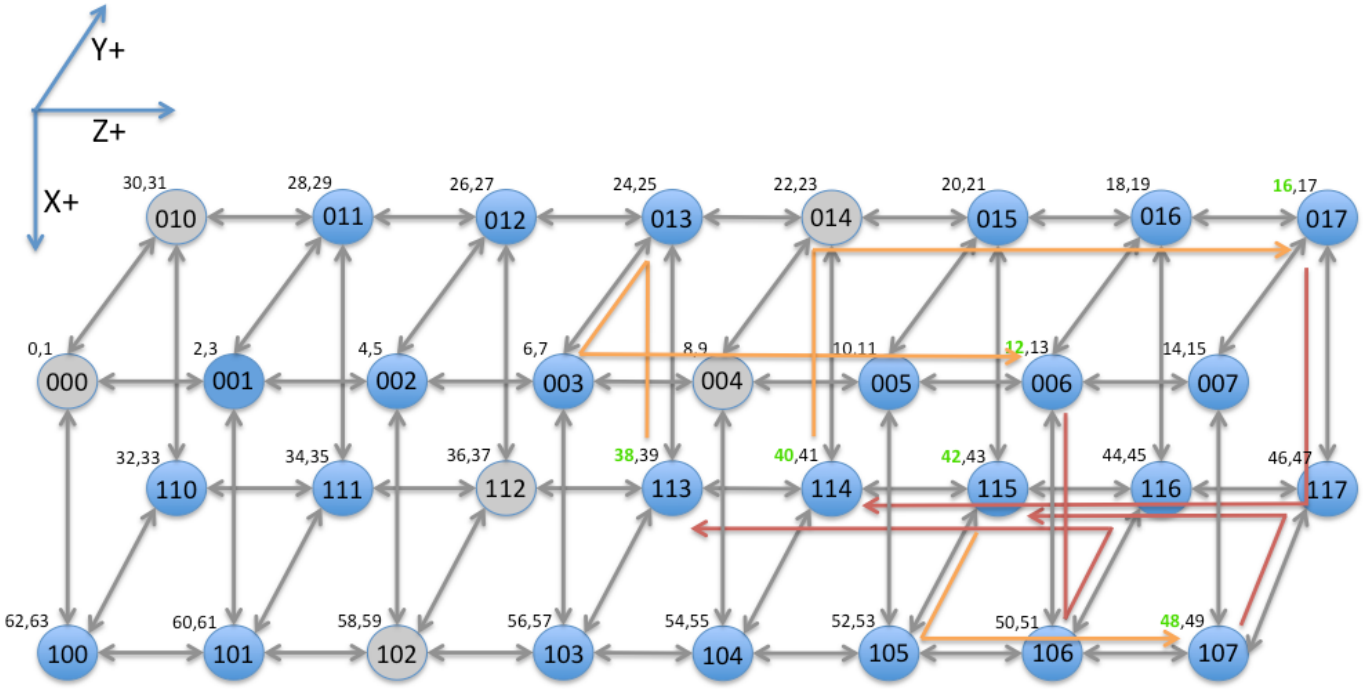

Fig. 1. Test system Architecture. Circles denote Gemini with arrows indicating traffic directions between Gemini. For clarity, torus-wrapping links are not shown. Grey circles are Gemini associated with service nodes, which are unavailable to applications. XYZ mesh coordinates label each Gemini. Two nodes sharing the same Gemini are indicated by pairs of node numbers. For example nid00016 and nid00017 share the Gemini at mesh coordinates 0,1,7. Traffic paths for the congestion traffic pairs are shown in orange and red.

III. Integration of Monitoring, Analysis, and Response Tools

Several components are needed to integrate system monitoring with resource- and architecture-aware task mapping: dynamic data collection and aggregation; incorporation of static routing information; interfaces to deliver the information to mapping tools; mapping tools that can use dynamic information; and applications that can accommodate mapping. We have developed a tool called the *ResourceOracle* (`RO`) that is the cornerstone of this integration.

Applications can obtain information directly from their local nodes (e.g., from `/proc/meminfo`, `/proc/stat`) to obtain coarse-grained understanding of how well they are utilizing their allocated resources. However, to gain insight into strategies to utilize allocated resources in a way that minimizes shared resource oversubscription, a global view is required. The *ResourceOracle* queries and aggregates platform-wide network-state information to provide such a global view. The `RO` also provides a user interface that enables applications or resource-management tools to easily obtain system information. The `RO` and its relationship with other integration components are described below.

A. Data collection and aggregation framework

In order to access and aggregate system-wide network data, we run the Lightweight Distributed Metric Service (LDMS) [18] monitoring and aggregation tool on all nodes. LDMS has samplers for relevant network data, and utilizes RDMA over Gemini to transport information from compute nodes to aggregator nodes in a low-overhead fashion.

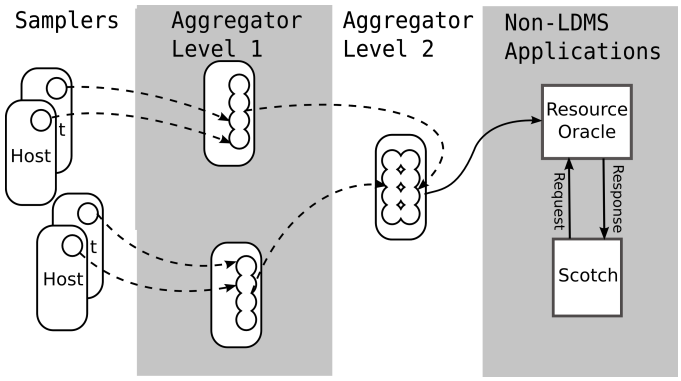

Fig. 2. High-level diagram of the framework components: LDMS monitoring and aggregation, ResourceOracle, and Scotch. Rounded rectangles denote LDMS daemons; circles within denote metric sets.

A high-level diagram of the LDMS monitoring and aggregation framework is shown in Figure 2. Samplers in this figure refer to hosts running monitoring daemons (one per host, including both compute and service nodes). Samplers collect information of interest on the hosts, including network performance counter information. The information is periodically pulled from sampler to aggregator or from aggregator to aggregator (depicted by dashed lines). The time interval is a user-specified configuration parameter; in this work, we used a two-second interval. Aggregators can also be queried by non-LDMS applications for their current data. Figure 2 depicts our *ResourceOracle* querying a second-level aggregator. For our small 64-node system, a single aggregator was used, as

LDMS supports fan-in from over ten thousand samplers to an aggregator.

A sample of the per-node information available from LDMS is shown in condensed form below. This information is the pertinent network information that can be used to infer network link congestion. Note that the metrics starting with X+ have counterparts in X-, Y+/-, and Z+/- directions which are not shown here.

```
U64 1  nettopo_mesh_coord_X
U64 1  nettopo_mesh_coord_Y
U64 6  nettopo_mesh_coord_Z
U64 13 X+_SAMPLE_GEMINI_LINK_USED_BW (%)
U64 0  X+_SAMPLE_GEMINI_LINK_INQ_STALL (%)
U64 0  X+_SAMPLE_GEMINI_LINK_CREDIT_STALL (%)
```

Mesh coordinates (example: nettopo_mesh_coord_X) are taken in conjunction to define the XYZ coordinates of each Gemini router (see Figure 1). USED_BW here provides the percentage of total theoretical bandwidth on an incoming link that was used over the last sample interval. INQ_STALL provides the percentage of time, over the last sample interval, that the input queue of the Gemini spent stalled due to lack of credits. CREDIT_STALL provides the percentage of time, over the last sample interval, that traffic could not be sent from the output queue due to lack of credits. These stalls are due to the Gemini network using credit-based flow control.

The percentage calculations are derived from link aggregated quantities of traffic counters (bytes) and time stalled (nanosec) presented by Cray's `gpcdr /sys` interface [22], which aggregates fundamental per-channel and NIC performance counter data [23]. Details of the percentage calculations can be found in [18]. USED_BW is proportional to the traffic per second on a link during the collection window as a fraction of the theoretical maximum bandwidth of the network media type. CREDIT(or INQ)_STALL is proportional to the time stalled divided by the number of lanes in the given direction normalized as a fraction of the total time between data samples.

B. Integration of system data with route information

In order to provide useful, global, network-related information to applicatons, we integrate collected data with full route information for all pairs of nodes. Applications can then query the `RO` for network information with the full route context.

Route information is built from the individual link information obtained from Cray's "`rtr --phys-routes`" command, while link-type information is obtained using the "`rtr --interconnect`" command. While the first produces a complete listing of pairwise routes including router tile information, the second produces a list of link directions. The second is also the source of the media type (independently used in the USED_BW calculation) which defines the maximum bandwith for that link. Limited output of each is shown below:

```
rtr --phys-routes:
23,24,33,34,43,44,53,54c0-0c0s0g000,01,10,11,25-27,35 ->
06,07,16-22,32c0-0c0s1g000,01,10,11,25-27,35 ->
06,07,16-22,32c0-0c0s2g000,01,10,11,25-27,35 ->
06,07,16-22,32c0-0c0s3g023,24,33,34,43,44,53,54

rtr --interconnect:
c0-0c0s0g0l00[(0,0,0)] Z+ -> c0-0c0s1g0l32[(0,0,1)]   LinkType: backplane
c0-0c0s0g0l01[(0,0,0)] Z+ -> c0-0c0s1g0l21[(0,0,1)]   LinkType: backplane
c0-0c0s0g0l02[(0,0,0)] X+ -> c0-0c1s0g0l02[(1,0,0)]   LinkType: cable11x
c0-0c0s0g0l03[(0,0,0)] X+ -> c0-0c1s0g0l03[(1,0,0)]   LinkType: cable11x
c0-0c0s0g0l04[(0,0,0)] X+ -> c0-0c1s0g0l04[(1,0,0)]   LinkType: cable11x
```

The nid number to cname string (e.g. c0-0c0s0n0) mapping is unique and can be found from, for example `/proc/cray_xt/cname` and `/proc/cray_xt/nid`. For example, `nid00012` is `c0-0c0s6n0` and is associated with Gemini `c0-0c0s6g0`. Given any two nodes, the entire route can be determined from this information. Also, the static path-length metric HOPS (the number links between any pair of nodes) can be computed from this route information.

The *ResourceOracle* is responsible for associating the dynamic monitoring information with the static system information. The `RO` parses the route information once, upon startup. On demand, the `RO` obtains the dynamic information for the relevant links from the current information in the aggregator (Figure 2). Bandwidth information is reported by the Gemini performance counters in the incoming direction. Thus, in evaluating, say, the maximum used-bandwidth in the route traversed between `nid00012` and `nid00038`, the contribution from the first hop is obtained from the value of the bandwidth used in the `X-` direction reported by `nid00050` (associated with the Gemini at mesh coords `(1,0,6)`) which is at the endpoint of the hop. Bandwidth and stall values for intra-node communication or for communication between nodes sharing the same Gemini are reported as 0.

C. Availability of the information to the mapping tools

The `RO` provides an interface to applications and, more specifically, to mapping tools by which they can retrieve the information of interest. Mapping tools require information that can be used for graph analysis in the mapping algorithms. The mapping tools identify nodes of interest by their nid numbers as obtained from `MPI_Get_processor_name`. Needed information includes coordinate information and link weighting information. Thus, the `RO` provides an API by which Gemini coordinate information and simple functions (e.g., min, max, sum) of well-known metrics (e.g., available bandwidth, credit stalls) of nodes and node pairs may be requested and returned. The `RO` listens on a socket for such requests. This interaction is depicted in Figure 2, where Scotch is receiving requested information from the `RO`.

D. Evaluation of information for use in mapping

The data available from the ResourceOracle then needs to be incorporated into models suitable for computing new task placements. Graph-based models are a natural choice for mapping, as graphs can be used to represent both the machine's network topology and the task dependencies of the application. Most tools for task placement rely on users to provide insight about the properties of the architecture or application. In this work, we use the Scotch graph mapping library [8], populating its graph models with dynamic information from the `RO`. Scotch accepts as input weighted graphs of the application tasks and of the machine topology. Function `SCOTCH_graphMap` then performs dual recursive bisection of both graphs to assign vertices of the task graph to vertices of the machine graph in a way that attempts to minimize the total cost of communication by the application.

In the application graph, each graph vertex represents one rank's computation. Task-graph edges represent the amount of data communicated between interdependent tasks. Edges exist only between tasks that share data.

The machine graph is the mechanism for providing LDMS state information to Scotch. Rather than represent the entire machine topology (i.e., all cores and links) in the machine graph, we build a graph representing only the allocated nodes. We construct a complete graph with one vertex per core; edges are weighted by the cost of sending data between the cores. Metrics from LDMS are incorporated into these graphs as edge weights. For example, with the HOPS metric, edge weights are the number of links in the routes between pairs of cores. Using the BW metric, an edge between two cores is weighted by the maximum percentage of bandwidth used along all links in the routes between the cores. With the STALLS metric, the edge weight is the maximum percentage of time spent in credit-stalls along any link in the edge's route. Graph edges between vertices i and j are undirected; we use the larger of the metric values from i to j and from j to i.

Scotch's mapping capabilities are serial at present. For these experiments, we gather the machine and task graphs to one core, compute the mapping, and broadcast the result. For moderate core counts, this solution is sufficient; future work will include parallel mapping strategies.

E. Re-mappable application

Finally, applications must be able to make use of the new task placement. Applications can be designed to dynamically migrate data to new cores when a new mapping is computed; this option allows a long-running application to adjust its task placement to accommodate the changing network conditions. Alternatively, mapping can be done as a pre-processing step to the actual computation, accounting, at least, for conditions at start-up time and avoiding the complications of data migration. For this paper, we chose the former approach, moving application data during execution to their new MPI ranks.

IV. Experimental Configuration

Our experiments evaluate the feasibility and benefit of remapping application tasks to cores based on run-time system data. We first show that competing traffic can negatively impact application run time. Then we show that we can obtain information about system traffic, and use that information to reduce application run time through preferential placement of communicating tasks on cores to avoid the competing traffic.

A. Sample Application

As a sample application, we chose an important computational kernel: sparse matrix-vector multiplication Ax. This kernel is used in a wide range of scientific applications. For example, it is the main component of iterative linear solvers used for finite element analysis, of eigensolvers used in structural mechanics, and of graph analysis algorithms such as PageRank. Our application, denoted SpMV, is built using the Trilinos [24] solver framework. Matrix and vector classes and operations are provided by Trilinos' Epetra package [25]. The matrix A generated in SpMV represents the discretization of Laplace's equation on a uniform grid. A has 8000 rows and 53,600 nonzeros, representing a $20 \times 20 \times 20$ grid. On 64 processors, each rank's submatrix represents a $5 \times 5 \times 5$ subgrid, resulting in a seven-point stencil-like communication pattern during matrix-vector multiplication. That is, ranks are arranged logically into a $4 \times 4 \times 4$ mesh, and each rank communicates with its north, east, south, west, front and back neighbors. Our x is a multivector of 100 vectors to which A is applied. Communication between neighboring ranks i and j consists of exchanging vector entries along the shared faces of the subgrids associated with i and j; thus, each message contains $5 \times 5 \times 100$ double-precision values. SpMV was run using 64 processes spread across the 16 allocated nodes, with four MPI ranks per node. Each experiment consisted of remapping tasks to cores, redistributing the data to reflect the new task assignment, and performing 10,000 matrix-vector multiplications. In the redistribution, the application migrated matrix and vector data among processors according to the new mapping; the MPI infrastructure (e.g., the MPI communicator) was not changed. All performance statistics shown are averaged over 24-44 experiments.

SpMV's task dependencies are modeled for Scotch in the task graph. Each task-graph vertex represents one rank's computation ($5 \times 5 \times 5$ subgrid and associated matrix and vector values). Task-graph edges represent the amount of communication between the tasks; for our experiments, these edges represent SpMV's seven-point stencil. Edges are weighted with the message size between processors: $5 \times 5 \times 100$ doubles. For our experiments, we used Scotch v6.0.0 [26].

B. Node Allocation and Competing Network Traffic

We generated competing background traffic using a simple bi-directional bandwidth benchmark. The benchmark continuously sends 1 MiB messages between two MPI processes as quickly as possible, placing load on the network links between the two processes. We place the two MPI processes on our test system based on the network topology and static routing algorithm to target specific network links. Multiple instances of the benchmark were placed on nodes with overlapping routes in order to increase the load on a single targetted link, as well as to create multiple hot spots in the network.

We created competing traffic between three pairs of nodes, as shown in Figure 1: `nid00016` and `nid00040`, `nid00048` and `nid00042`, and `nid00012` and `nid00038`. These nodes are labeled in green; forward and reverse communication routes are shown in red and orange, respectively. Significant traffic is routed between Gemini at mesh coords (1,1,5) and (1,1,4), (1,1,6) and (1,1,5), and (1,1,7) and (1,1,6). The maximum percentage of time spent in credit stalls along any link induced by this competing traffic is roughly 68% on the `Y+` link out of Gemini (1,0,6), with that out of Gemini (1,0,7) being comparable. The maximum percentage of available bandwidth used along any link is 61% on the `Z+` link into Gemini (1,1,5). The maximum theoretical bandwidth in the `Y` direction is less than that in the `X` and `Z` directions, resulting in the potential for a higher percentage of time spent in stalls in the `Y` direction, even for equivalent traffic. These generated values are in line with values seen during production conditions on Blue Waters. In a representative dataset of a day's worth of data presented in [18], several instances of values of 60+% for time spent in credit stalls occurred over time ranges of up to 1.5 hours, with values of 30-40+% ranging over 20 hours for some links. The percentage of bandwidth used was not considered in [18], but values of 60+% can be observed in that same dataset.

These values were determined over a collection interval of one minute.

Application nodes were chosen to include those that could potentially route traffic through congested links, depending on where the tasks are placed. Application nodes are shown in Figure 3 labeled in red (`nids 17-19,24,32-35,39,41,43-45,49-51`), as are their associated Gemini. There are then a total of 256 unique routes between the 16 nodes. The distribution of HOPS for the possible routes in the application is shown in Figure 4 (top); the distribution of representative values of STALLS and BW for the possible routes due to the competing traffic is shown in Figure 4 (bottom). Task placement determines which routes are actually utilized during execution. The goal of remapping is to minimize the use of links involved in the congested paths in Figure 1.

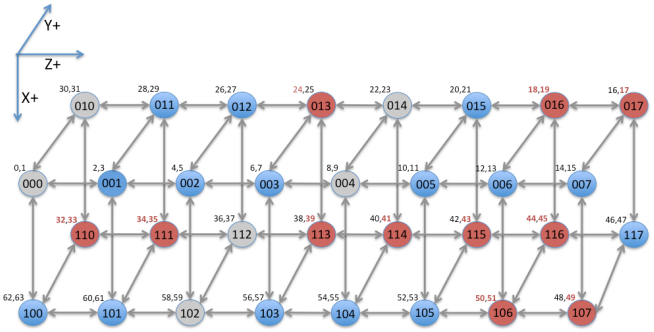

Fig. 3. Node allocation for our experiments. Nodes allocated to the application and their associated Gemini are shown in red. Edges designate hardware links.

C. LDMS

The LDMS samplers were configured to sample a variety of data, including the High-Speed Network performance counters and Gemini grid coordinates on all nodes (compute and service), at intervals of two seconds. An aggregator was configured to collect data from the samplers at the same two-second interval. Samplers were configured to sample synchronously. The aggregator was configured to collect from the samplers approximately 100ms after sampling. This synchronization between samplers provides a *system snapshot* of the sampled state, while the aggregator's delay ensures that samples are complete when collected. Both raw counter data and derived values based on the counter values over the last interval (e.g., percent time spent in stalls) were made available in the data set, and, thus, were accessible via the `RO`. The data used in mapping decisions was based on the derived data at the time of the query; thus, mapping decisions were based on a two-second average behavior of the chosen metric values that were up to two seconds old. The data associated with aggregation is 1088 bytes per sampler or 68kB overall every two seconds and, thus, does not contribute significantly to network traffic or congestion. No historical values are retained by either the samplers or aggregator beyond their current data sets.

V. Experimental Results

Our first experiment demonstrates that our congestion utility does, indeed, generate enough interference to affect SpMV

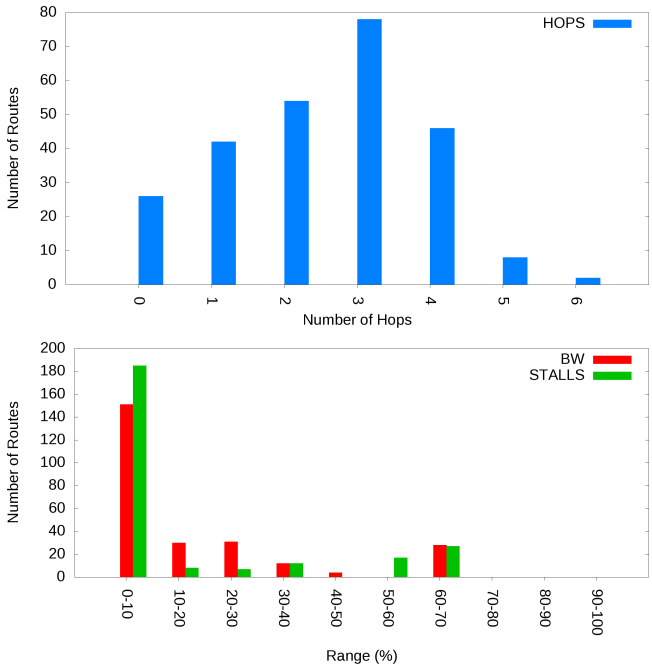

Fig. 4. Impact of network geometry and of competing traffic on the potential application routes. Distribution of HOPS values for the possible application routes are shown at the top. Distribution of representative values of STALLS and BW for the possible application routes due to the competing traffic are at the bottom. (Range values are inclusive of the low and exclusive of the high.) There are 256 possible unique routes. Task placement determines which routes are actually utilized during execution.

	Average SpMV time (secs) for 10K matvecs
Without Congestion	5.07
With Congestion	6.03

TABLE I. Execution time of SpMV without and with congestion, demonstrating the effectiveness of the congestion utility.

execution time. We first measure the execution time of SpMV with no congestion in the network. Then we launch three instances of the congestion utility to generate network traffic as shown in Figure 1, and rerun SpMV. The execution times with and without congestion are shown in Table I. The congestion utility increases the average execution time by 18.9%.

Next we apply Scotch remapping to SpMV in the congested environment. We compute new task placements based on the metrics HOPS, BW, and STALLS. To show that our metrics are important to finding a good task placement, we also compute a new map using no metric. In this "No Metric" case, we provide a complete architecture graph with uniform weights to Scotch. We again measure the execution time of SpMV. For each mapping, we compute the estimated communication cost of each message in SpMV as the product of the number of bytes in the message and the machine-graph edge weight for the route taken by the message. This total estimated communication cost is the metric Scotch tries to reduce through remapping.

In Figure 5 (left), we show the maximum estimated communication cost per message for each mapping metric, as well as the base case with the default MPI placement of tasks on cores (i.e., no mapping). Results are normalized with respect

to the "no mapping" results. We see that Scotch is effective in reducing the estimated communication costs for the HOPS, BW, and STALLS metrics. Because the "no metric" case uses uniform machine-graph weights, the estimated communication cost is identical to the "no mapping" case.

In Figure 5 (right), we show the average execution time for 10,000 matrix-vector multiplications in SpMV. The solid black line represents the normalized execution time of SpMV without congestion; this time is the ideal execution time we could achieve if the mapping could completely avoid the congestion. As expected, we see that mapping with no metric increases execution time; in this case, the uniformly weighted machine graph incorrectly tells Scotch that all routes are equally good, so Scotch can place tasks arbitrarily in the machine. The static HOPS metric offers some improvement (execution time reduced 2.2%), as locality of tasks within nodes is enabled. However, this metric does not account for congested links, so limited benefit is seen. The BW and STALLS metrics offer greater benefit than HOPS, reducing the execution time by 6.9% and 7.8%, respectively.

Figure 6 shows, for each mapping method, the percentage of the extra execution time due to congestion that is recovered by performing mapping. Here, higher values are better, indicating that more of the congestion overhead is alleviated. Again, we see that the dynamic BW and STALLS metrics provide the greatest recovery from the congestion overhead. "No Metric" results are not included, since mapping with no metric resulted in higher execution times.

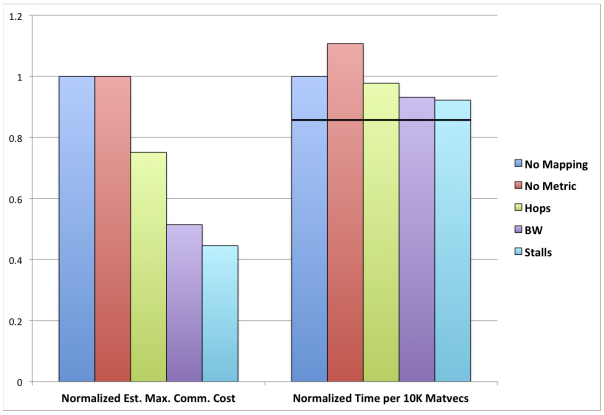

Fig. 5. Estimated maximum communication cost per message (with respect to the machine graph used in mapping) and actual execution time for 10K matrix-vector multiplications using no mapping, mapping with no metric, and mapping with the HOPS, BW, and STALLS metrics. All results are normalized to the "no mapping" case. Lower values are better, indicating lower estimated communication and faster execution. The solid line in the right figure indicates the "ideal" time: the execution time without congestion.

As a final test, we compare our mapping methods with the geometric mapping technique of Deveci et al. [6]. Their method *MJ* computes mappings based on the physical proximity of cores and tasks, rather than on their graph connectivity; physical proximity serves as a proxy for interdependence of tasks. *MJ* applies geometric partitioning to both the Gemini coordinates for each core and to geometric coordinates representing each task's data. For our experiments, we use the average grid coordinates for each task's $5 \times 5 \times 5$ subgrid as the task's geometric coordinates. The geometric partitioner

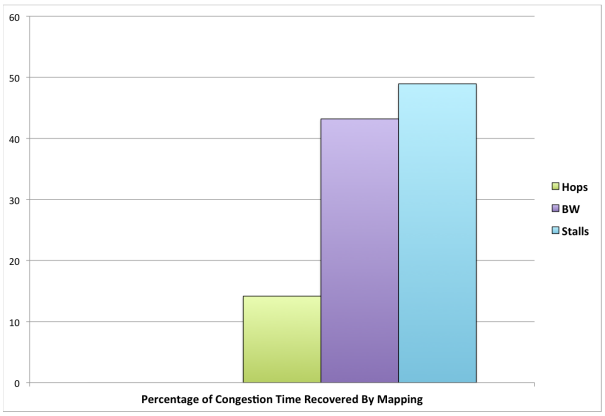

Fig. 6. The percentage of congestion time recovered by performing mapping. Higher values are better, indicating that more of the overhead caused by congestion is alleviated by mapping.

	HOPS	BW	STALLS	MJ
Percentage of Congestion Time recovered	14.2	43.2	48.9	25.5

TABLE II. PERCENTAGE OF CONGESTION TIME RECOVERED THROUGH GRAPH-BASED MAPPING WITH STATIC HOPS AND DYNAMIC BW AND STALLS METRICS, AND WITH STATIC GEOMETRIC MAPPING *MJ* [6].

assigns each task and core to a part; tasks and cores in the same part are mapped to each other. Currently, *MJ* is unable to use information about stalls or bandwidth, although one could consider scaling the Gemini coordinates based on this dynamic information. Since the Gemini coordinate information does not change due to congestion in the network, we expect *MJ* to produce results most similar to graph-based mapping with the static HOPS metric. Indeed, in Table II, we see that static geometric mapping with *MJ* recovers roughly 25.5% of the congestion time, a bit more than graph-based mapping with HOPS, but less than graph-based mapping with dynamic metrics BW and STALLS.

Given that this paper is meant to demonstrate the potential impact of using dynamic monitoring information for resource assignment through mapping, we have focused on the benefit of resource-aware task placement for our application. However, the cost of performing the remapping is also important; our future work will address reducing the cost of remapping. In our experiments, the time required for computing a new map (including serial Scotch mapping and broadcast of the results) averaged 0.012 seconds. The cost for actually redistributing the matrix and vector data to implement the new map was also small, averaging 0.004 seconds.

The bulk of the overhead was incurred in obtaining the dynamic data, which depends on mechanics of the `RO` which are not yet optimized. To remain fully generic, we intentionally did not integrate the routing analysis with the data-collection system. The interface of the `RO` with generic system monitoring requires user-space calls and processing which could incur meaningful overhead. Our future work includes integration of the `RO` with our LDMS daemon so that the `RO` reads the required data directly from the internal data structures and, thus, incurs negligible overhead for obtaining the dynamic data. We can get an upper bound on the overhead after this integration by considering the cost of obtaining the HOPS

metric in our current configuration, since the HOPS metric uses static data and, thus, bypasses the user-space call for obtaining the dynamic data but is the same in all other respects. For the HOPS metric, the time to obtain the data for one mapping is 0.27 seconds, on average. Overhead could also be reduced by improving our mapping interface to the `RO` to bundle several requests into a single query.

VI. Conclusion and Future Work

We have shown the potential for dynamic monitoring to improve the performance of parallel applications in shared environments. By integrating data collection from LDMS, data aggregation in the *ResourceOracle*, and graph-based task placement in Scotch, we demonstrated the ability to avoid congestion in shared networks, reducing application execution time for a key computational kernel, SpMV. Our work shows that utilizing global knowledge of network state can significantly improve application performance. It is a first step toward a scalable integrated system for very large-scale simulations.

Still, several challenges remain. From related work [5], [6], we expect even greater benefit from dynamic mapping in large-scale applications. Experimentation with our system at large scale and in a production environment is needed. Further analysis and reduction of our system's overhead are also needed, particularly at scale. The major reduction in overhead will be through enabling the `RO` to have direct access to the LDMS Aggregator's data. Greater scalability of our approach might be achieved through the use of multiple, distributed `RO` as well as batching `RO` queries from the mapping tools. Serial mapping is sufficient for O(100K) tasks and nodes, but at larger scale, parallel mapping algorithms may also be needed. We will also evaluate more closely which system metrics are of most importance for parallel performance, and their sensitivity to relative weighting and transient phenomena. Supporting time-windowed data in the `RO` will decrease the likelihood that application responses are overly sensitive to small or short-term environmental changes. Finally, we will pursue use of integrated monitoring and finer-grain response at the application level through data-level dynamic load balancing.

Acknowledgment

The authors thank Jason Repik (Cray, Inc.) for useful discussions on congestion in the Cray XE platform and for assistance in running the experiments in this work. They also thank Siva Rajamanickam, Vitus Leung, Stephen Olivier, Andrey Prokopenko, Erik Boman, Mehmet Deveci and Umit Catalyurek for helpful discussions and feedback.

References

[1] A. Bhatele, K. Mohror, S. Langer, and K. Isaacs, "There Goes the Neighborhood: Performance Degradation due to Nearby Jobs," in *Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, 2013.

[2] M. Bender, D. Bunde, E. Demaine, S. Fekete, V. Leung, H. Meijer, and C. Phillips, "Communication-aware processor allocation for supercomputers," in *Algorithms and Data Structures*, ser. Lecture Notes in Computer Science, F. Dehne, A. Lpez-Ortiz, and J.-R. Sack, Eds. Springer Berlin Heidelberg, 2005, vol. 3608, pp. 169–181.

[3] C. Albing, N. Troullier, S. Whalen, R. Olson, and J. Glensk, "Topology, bandwidth and performance: A new approach in linear orderings for application placement in a 3D torus," in *Proc Cray User Group (CUG)*, 2011.

[4] J. Enos, "Application Runtime Consistency and Performance Challenges on a Shared 3D Torus." MoabCon 2014, 2014. [Online]. Available: http://www.youtube.com/watch?v=FR274JitRq8

[5] R. Barrett, C. Vaughan, S. Hammond, and D. Roweth, "Reducing the Bulk of the Bulk Synchronous Parallel Model," *Parallel Process Lett*, vol. 23, no. 4, 2013.

[6] M. Deveci, S. Rajamanickam, V. Leung, K. Pedretti, S. Olivier, D. Bunde, U. V. Catalyurek, and K. Devine, "Exploiting Geometric Partitioning in Task Mapping for Parallel Computers," in *Proc. 28th Int'l IEEE Parallel and Distributed Processing Symposium*, 2014.

[7] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proc 25th Intl Conf Supercomputing*. ACM, 2011, pp. 75–84.

[8] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking*. Springer, 1996, pp. 493–498.

[9] C. Walshaw and M. Cross, "Multilevel mesh partitioning for heterogeneous communication networks," *Future Generation Comp Syst*, vol. 17, no. 5, pp. 601–623, 2001.

[10] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - 18th Euromicro Int'l Conf Parallel, Distributed and Network-Based Computing*, IEEE, Ed., 2010.

[11] T. Agarwal, A. Sharma, and L. V. Kalé, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *Proc. Int'l IEEE Parallel and Distributed Processing Symposium*, 2006.

[12] A. Bhatelé and L. V. Kalé, "Benefits of Topology Aware Mapping for Mesh Interconnects," *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)*, vol. 18, no. 4, pp. 549–566, 2008.

[13] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *Proc. 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 21–.

[14] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[15] "Ganglia." [Online]. Available: http://ganglia.info

[16] "Nagios." [Online]. Available: http://nagios.org

[17] "Performance Co-Pilot Programmer's Guide," March 2014. [Online]. Available: http://oss.sgi.com/projects/pcp/doc/pcp-programmers-guide.pdf

[18] M. Showerman, J. Enos, J. Fullop, P. Cassella, N. Naksinehaboon, N. Taerat, T. Tucker, J. Brandt, A. Gentile, and B. Allan, "Large Scale System Monitoring and Analysis on Blue Waters using OVIS," in *Proc. Cray User's Group*, 2014.

[19] National Center for Supercomputing Applications, "Blue Waters." [Online]. Available: https://bluewaters.ncsa.illinois.edu

[20] "The Gemini Network," Aug 2010. [Online]. Available: http://wiki.ci.uchicago.edu/pub/Beagle/SystemSpecs/Gemini_whitepaper.pdf

[21] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *Proc. 2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2010.

[22] "Managing System Software for the Cray Linux Environment," Cray Inc. S-2393-4202, Tech. Rep., 2013.

[23] "Using the Cray Gemini Hardware Counters," Cray Inc. S-0025-10, Tech. Rep., 2010.

[24] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM TOMS*, vol. 31, no. 3, pp. 397–423, 2005.

[25] "Epetra Linear Algebra Services Package." [Online]. Available: http://trilinos.sandia.gov/packages/epetra

[26] F. Pellegrini, *SCOTCH 6.0 User's Guide*, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Dec 2012.