The Rails Toolkit – Enabling End-System Topology-Aware High End Computing

Venkatram Vishwanath, Jason Leigh,
Sungwon Nam, Luc Renambot
Electronic Visualization Laboratory
University of Illinois at Chicago,
Chicago, Illinois, USA
venkat@evl.uic.edu

Takashi Shimizu, Hirokazu Takahashi,
Makoto Takizawa, Osamu Kamatani
NTT Network Innovation Laboratories
Yokosuka, Japan
t-shimizu@ieee.org

Abstract— **We present a novel rails approach so that future e-Science applications can effectively exploit future system architectures, including multi-core and many-core architectures, multiple network cards, multiple graphical processing units and hybrid hierarchical memory architectures. We define "rail" as the co-scheduling of two or more of these resources. This approach enables creation of parallel multi-rails through every aspect of an end system: from processing on the multi- and many cores, to generation of multiple data flows and streaming over multi-lane NICs connected via a parallel interconnect. We describe a novel open-source multi-rail toolkit and evaluate end-system parameters that impact the efficiency of such multi-rail systems, including Interrupt, Memory, Thread and Core Affinities -- key properties for achieving scalable performance.**

Keywords- High-performance computing, topology-aware resource allocation, high-level abstractions, multi-core computing

I. INTRODUCTION

Cyberinfrastructure, comprised of geographically distributed instruments and compute, storage and visualization clusters interconnected by ultra-high-speed optical networks, is the technological foundation that enables global e-Science research, in fields including climate analysis, astronomy, high-energy physics and neuroscience. Earth scientists in National Aeronautics and Space Administration (NASA) climate modeling, analysis and prediction (MAP) project [1] routinely run simulations and models on geographically distributed computational clusters and access distributed storage to predict tropical cyclones and hurricanes. Bio-scientists and geo-scientists are adopting the Scalable Adaptive Graphics Environment (SAGE) [2], specialized collaboration and visualization middleware that enables co-operative scientific discovery by geographically distributed scientists.

e-Science applications have demonstrated scalable performance using today's cyberinfrastructure architectures. However, future architectures, as listed in Table 1, will require today's applications and middleware to scale their performance in ways previously unexplored. Future cyberinfrastructure will be characterized by deep and complex memory, processor and interconnect hierarchies with inherent parallelism in the various subsystems. Thus, a critical component for scalable performance will be the development of new and novel techniques for efficient utilization of end-system architectures and resources.

Typically, e-Science applications and middleware scale their performance to end-systems by optimizing their implementations for the end-system architecture. However, as end-system architectures evolve and become more complex, solutions that aid in the design of evolvable software are of paramount importance. One way to achieve this would be to develop abstractions of the various subsystems. These abstractions can help e-Science programmers design efficient and deployable middleware and applications.

We present the Rails Toolkit (RTK), an approach towards enabling e-Science applications and middleware to effectively exploit the potential of these architectural trends. RTK abstracts end-system topology for applications and middleware, and enables co-scheduling of CPU cores, GPUs, memory and network resources within multi- and many-core

TABLE I. ARCHITECTURAL TRENDS THAT E-SCIENCE APPLICATIONS AND MIDDLEWARE WILL HAVE TO CONTEND WITH IN FUTURE

Subsystem	Currently Deployed Architecture	Future Architectural Trends
Processor	Dual and Quad core	Multi- and Many-cores with a Multi-dimensional topology
Memory	SMP, NUMA (typically 2 memory banks)	SMP, NUMA, Hybrid combination of SMP and NUMA, Multi-dimensional (2D and 3D) memory topology
Graphical Processing Unit (GPU)	PCIe based GPU (typically with 128 processors)	Multiple GPUs with 256 to 800 processors per GPU (potentially on-core GPU design)
System Interconnects	Shared Bus, PCIe Gen 1 (2.5 Gbps)	Multi-lane PCIe Gen 2 and 3, Quick processor interconnect (QPI), HyperTransport, (HT) DWDM-based optical interconnects
Network Interconnects	10 GE Ethernet, Infiniband, Myrinet, etc.	40 Gbps – 100 Gbps Multi-lane Ethernet, Infiniband interconnects, Multi-lane DWDM based interconnects
Wide-Area Network	1-10 Gbps networks	DWDM-based Multi-10 Gbps optical networks

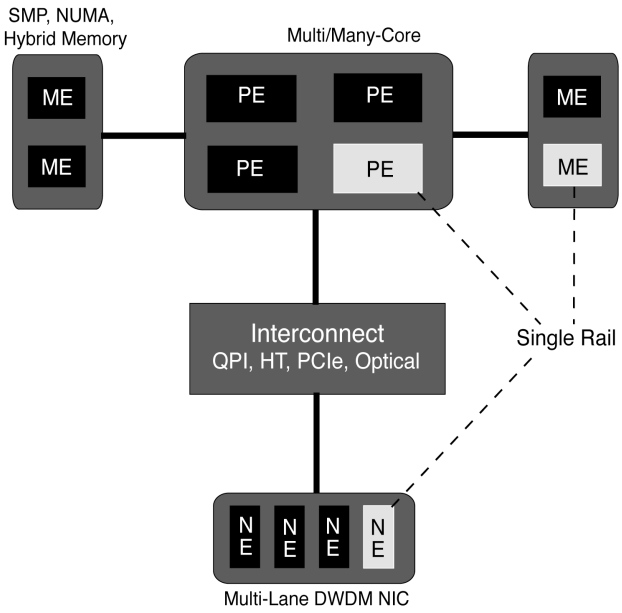

Figure 1. The Rails approach. This figure depicts a network rail wherein co-allocation of memory elements (ME), processor elements (PE) and networks resources (NE) help achieve improved performance

computer systems. We define a "rail" as the co-scheduling of two or more of these resources. Using RTK, application developers can create one or more rails over which their data-intensive computations and data retrievals can be accelerated with minimal interference from other rails or applications, and thus dramatically improve program performance. RTK is an open source toolkit and presents an intuitive API for applications and middleware to efficiently utilize end-system architectures. RTK can be used to improve the performance of high-performance computing applications, high-speed data delivery applications, and high-resolution graphics and video streaming. In the case of SAGE, RTK can aid in the efficient use of network parallelism to improve the performance of data streaming critical for real-time collaboration.

Novel contributions of the paper include:

- End-system, topology-aware, resource abstractions to enable applications efficiently utilize current and future high-end systems. This enables topology-aware memory allocation and enables applications to seamlessly use multiple network interfaces.
- Easy-to-use extensible API that can be integrated at multiple levels with applications and middleware.
- Open source application-level toolkit available for download for immediate use by the community.

The outline of the paper is as follows: We describe the rails approach and RTK Toolkit in section II. Experimental evaluation using the RTK Toolkit, including micro-benchmarks and application-level benchmarks, is discussed in section III. We present related work in section IV and finally conclude in section V.

II. RAILS TOOLKIT (RTK)

e-Science applications typically have multiple components, which are compute-intensive, network-intensive, IO-intensive and combinations of these. In network-intensive workloads, including bulk data transfer and data streaming, improving the achievable throughput and reducing the message latency is critical for performance. Additionally, reducing message latency and memory access latency is important for compute-intensive e-Science applications. The Rails approach enables efficient topology-aware co-allocation of system resources, including memory, processing cores and network subsystem resources. Figure 1 depicts a network rail which is a software abstraction of a processor core connected to a lane on a network interface card (NIC) via a dedicated interconnect. A network rail helps in improving the achievable throughput and reducing the message latency by reducing cache pollution and lowering memory access latency. The rails approach enables pipelining of multiple subsystems to compose hybrid rails. The RTK API can be used to pipeline GPU rails and network rails, and thus improve the performance of graphics streaming for remote visualization by reducing cache pollution, exploiting memory locality to reduce latency and reducing system bus contention. This is critical for future cyberinfrastructures where GPUs are an integral component. RTK enables allocation of parallel rails, which, facilitates exploitation of system topology and the parallelism inherent in current (and future) system architectures. A parallel four-rail network rail system is depicted in Figure 1, each rail consists of a processor core with dedicated memory connected to a lane on a NIC via a dedicated interconnect. The parallel rails approach can be expanded to exploit parallelism in other sub-systems.

RTK is implemented in C++ and is distributed under GNU Public License (GPL) version 2.1. A beta version of the toolkit and relevant documentation is available for download at http://www.evl.uic.edu/cavern/rtk. It works under Linux and has been tested on SMP-based Intel architectures, NUMA-based AMD Opterons and IBM Cell architectures. We describe the Rails toolkit architecture in section II-a, discuss system properties which help in achieving topology-aware resource allocation in section II-b and provide an example of the RTK API in section II-c.

A. Rails Toolkit Architecture

Figure 2 depicts the Rails Toolkit Architecture, which consists of the Resource Abstraction Layer, Resource Allocation Layer and the Rail Allocation Layer. The **Resource Abstraction Layer** abstracts the end-system topology and deals with the low-level resource bindings. The **Resource Topology Database** maintains relevant information including the topological configuration of the available processors, cores, memory nodes and IO devices. This database is populated during initialization by probing the system resources and using input configuration files. The **Resource Binding Layer** is responsible for binding interrupts to processor(s), threads to processor(s), the

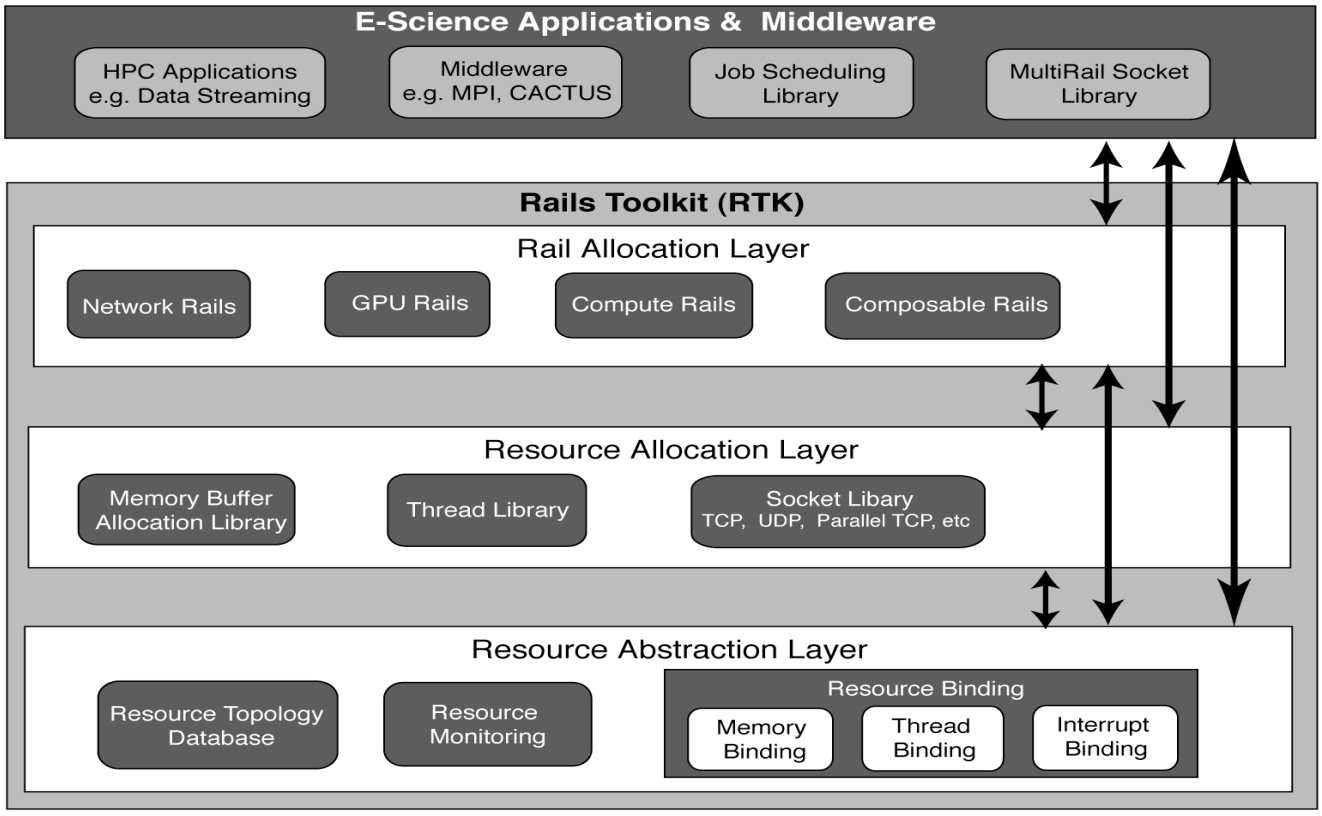

Figure 2. Rail Toolkit Architecture. This figure depicts the various layer in the design of the Rail Toolkit (RTK) architecture. RTK consists of Resource Abstraction Layer, Resource Allocation Layer and Rail Allocation Layer,. E-Science applications and middleware can use any of the 3 layers for optimizing their performance to an end-systems topology.

memory policy and allocation over node(s). This layer is designed using a wrapper around Linux system calls and enables co-allocation of the resources. The **Resource monitor** is a lightweight daemon that periodically checks the online status of the processors and memory nodes.

The **Resource Allocation Layer** allocates threads, sockets and memory using the underlying resource abstraction layer. The **Thread Library** is a C++ wrapper around the pthread library and enables manipulation of the processor-thread binding and memory policies of a thread using the resource abstraction layer. Additionally, it provides in-depth performance statistics, including context-switches and priorities, on a per-thread basis. The **Memory Allocation Library** enables topology aware memory allocation. It supports the NUMA memory policies available in Linux including interleaving, local allocation and strict allocation. The **socket library** currently supports TCP, UDP and Parallel TCP. The library is extensible and is useful in the design of composable protocols such as Reliable Blast UDP [3] and LambdaStream [4]. The library provides in-depth performance information of the network streams.

The Rail Allocation Layer synergistically co-allocates resources for improved performance. This layer can aid in isolating resources and reducing contention. The layer also enables pipelining of rails. In graphics streaming applications, pipelining GPU and network rails is important for reducing resource contention, including the contention in IO bus due to the GPU and network subsystem competing for it.

We have exposed the capabilities at various layers as a lightweight API so that researchers interested in applying this approach have multiple levels to integrate their applications with RTK. Details of the API are available at the RTK website. We envision middleware and applications using RTK to fully exploit the topologies of end-systems. RTK could be used in the design of adaptive run-time systems to optimize resource allocation. One such example is the **MultiRail Socket Library**, which enables seamless use of multiple network rails for network intensive applications. It is implemented in C++ and derived from the rail socket library. It exploits:

- Data Parallelism by striping the data onto multiple data streams. The current implementation uses static data striping and can be augmented to use adaptive striping policies [5].
- Task Parallelism by employing worker threads to stream each of the data streams. This makes efficient use of system resources in a multi-core, many-core environments.
- Network Parallelism by streaming the data streams over the multiple network paths available between a source and destination pair.

Additionally, the library employs efficient memory interleaving heuristics to improve memory bandwidth.

B. *System properties critical for topology-aware resource allocation*

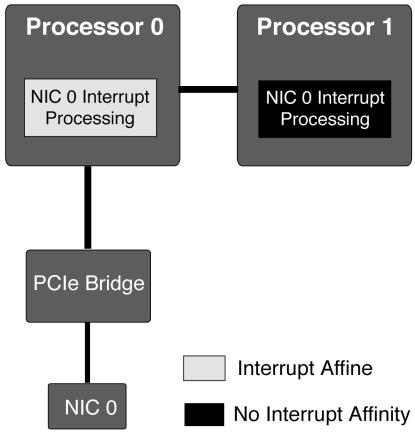

Figure 3. This figure depicts the Interrupt Affinity (IA) property. IA is set if the interrupt processing occurs on a processor where the device is physically connected

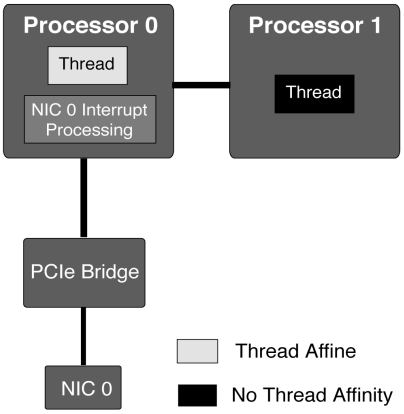

Figure 4. This figure depicts the Thread Affinity (TA) property. TA is set if the network application thread is bound to the processor where the interrupt processing occurs

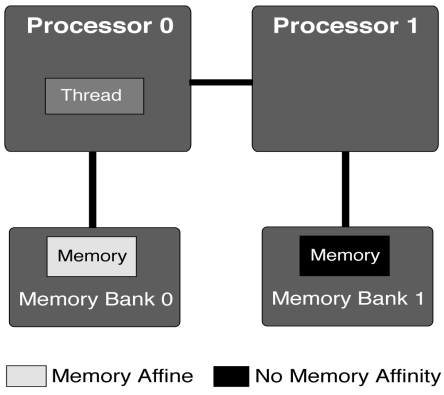

Figure 5. This figure depicts the Memory Affinity (MA) property. MA is set if the application buffer is allocated on the memory bank where the network application thread is bound.

We discuss properties that help improve an application's performance by enabling efficient topology-aware resource allocation. In this paper, we restrict our focus towards properties critical for network-intensive workloads. However, we would like to note that these properties are also necessary for other e-Science workloads, including compute-intensive workloads.

We define the ***Interrupt Affine*** **property** as one wherein the interrupt processing is performed on the processor to which the IO device is physically bound. Interrupt affinity reduces the message latency by servicing the interrupts on the nearest processor. As seen in Fig. 3, NIC 0 is physically attached to the PCIe bridge physically connected to processor 0. If the interrupt processing of NIC 0 occurs on any core of processor 0, we consider this to be interrupt affine. If the interrupt processing of NIC 0 occurs on processor 1, the interrupt affinity is not set. Thus, we define interrupt affinity relative to the physical topology of the IO device.

We define the ***Thread Affine property*** as one wherein the network application thread is scheduled on the processor in charge of the interrupt processing. Thread affinity reduces cache pollution and improves latency as the interrupt processing and the application thread are scheduled on the same processor. As seen in Figure 4, if the network application thread is scheduled on processor 0 and the interrupt processing of NIC 0 occurs on processor 0, we consider this to be thread affine. In Figure 4, we have both interrupt affine and thread affine properties. Thus, for network intensive workloads, we define thread affinity relative to the corresponding interrupt processing.

We define the ***Memory Affine Property*** as one wherein the memory buffer used by the network application thread is allocated on the memory bank with the lowest access latency with respect to the application thread. In case of NUMA-based systems, memory allocation on the local memory bank is considered to be memory affine. In Figure 5, the network application thread is scheduled on processor 0. If the memory is allocated on node 0, we consider this to be memory affine. Memory affinity helps in reducing the data access latency. In case of system architectures with deep and multi-level memory hierarchies, memory affinity refers to allocation of memory on memory nodes with the least access latency. Lower memory access latency is critical for data-intensive e-Science.

Additionally, using RTK, one can enable multiple properties simultaneously for improved performance. Enabling thread and interrupt affinity together would help in an improved performance over the individual affinities due to lower cache pollution among others. As mentioned earlier, RTK can be used to form parallel rails to exploit the inherent parallelism in end-systems.

If **T** is the achievable performance of a single rail, In an **N**-rail system, the expected performance would be: **N x T x** ∂, where ∂ is the parallel efficiency. In an ideal parallel system, the parallel efficiency is approximately unity ($\partial \rightarrow$ **1),** and this system exhibits additive performance. The goal would be to identify the affinity combinations that would

TABLE II.
PSEUDOCODE OF A SIMPLE TCP CLIENT

```
int sockfd; struct sockaddr_in server_addr;

/* create a TCP socket */
sockfd=socket(AF_INET,SOCK_STREAM, 0)

/* Populate the server_addr with server info */

/* connect to the server */
connect(sockfd,&server_addr,sizeof(srv_addr))

/* Send data to the server */
loop{
      write ( sockfd, buffer, sizeof(buffer));
}

close (sockfd);
```

TABLE III.
TCP CLIENT USING RTK'S RAIL ABSTRACTION LAYER API

```
int sockfd; struct sockaddr_in server_addr;

/* create a TCP socket */
sockfd = socket(AF_INET,SOCK_STREAM,0)

/* Populate the server_addr with server info */

/* connect to the server */
connect(sockfd,&server_addr,sizeof(srv_addr));

/* Initialize Interrupt Affinity of the network
* interface to its physically connected processor
* and set thread affinity to the processor where
* the Interrupt affinity is set */
RTK_Interrupt_Affinity IntrAff;
RTK_Thread_Affinity ThreadAff;

IntrAff.init(sockfd);
ThreadAff.init ( &IntrAff);

/* Send data to the server */
loop {
      write ( sockfd, buffer, sizeof(buffer));
}
close (sockfd);
```

TABLE IV.
TCP CLIENT USING RTK'S RAIL ALLOCATION LAYER API

```
RTK_tcpClient client; /* A TCP Client Object */

/* connect to the server */
client.connectToServer(srv_name, srv_port);

/* Initialize Interrupt Affinity of the network
* interface to its physically connected processor
* and set thread affinity to the processor where
* the Interrupt affinity is set */

client.setInterruptAffinity();
client.setThreadAffinity();

/* Send data to the server */
loop{
     client.write ( buffer, sizeof(buffer));
}

client.close();
```

help parallel efficiency. This could be used towards the design of efficient run-time systems.

C. *Sample program using the RTK API*

Table II depicts the pseudo code of a typical TCP client socket program. This program uses the default affinity settings of the operating system. A rail-enabled socket program using the RTK API to enable interrupt affinity and thread affinity given in Table III. This uses the RTK Abstraction Layer API. One could design an interposer layer to set these affinities by default for e-Science applications. The interposer layer enables easy integration of the RTK with existing applications without a single line of source-code modification. Table IV demonstrate the same using the RTK Allocation layer. This provides an intuitive higher-level abstraction for resource allocation. In-depth examples, including querying the system topology and multi-node memory allocation, can be found in the RTK software distribution

III. EXPERIMENTAL ANALYSIS

In this section, we study the efficacy of the RTK toolkit on a set of micro-benchmarks and application-level benchmarks. We focus our attention on network-intensive benchmarks. The experimental testbed consisted of Two dual-core, dual-processor AMD 2.6 GHz Opteron TYAN 2895 systems with 4GB RAM and two PCIe 16X slots. The two machines were connected back-to-back with two 10 GE Myrinet NIC each. The Linux kernel version used was 2.6.18 with MSI enabled. The MTU used for the experiments was 9000 bytes. The 1.4.1 Myrinet driver was used in the experiments

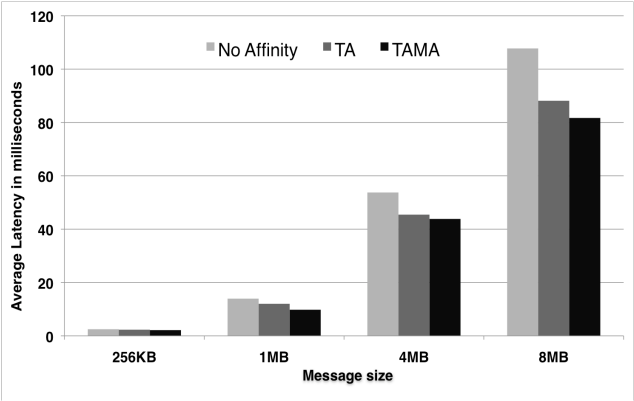

Figure 6. Effect of affinities on the latency of multiple TCP streams. As the message size increases, memory affinity is key for improved throughput

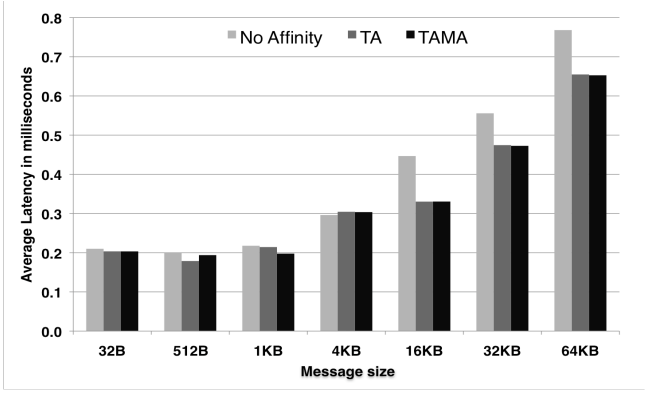

Figure 7. Effect of affinities on the latency of multiple UDP streams. Thread affinity plays a key role in inmproved throughput

A. Micro-benchmarks

We evaluate the performance of rails and the RTK toolkit on network intensive workloads. In e-Science cluster-based applications, a node routinely needs to send and receive data from multiple nodes. We designed a simple TCP micro-benchmark program to measure the efficacy of the rails approach on the achievable throughput, CPU usage and message latency. The benchmark program is written in C and creates four network streams between the two test nodes. Two network streams are bound to each of the 10G NIC. The client and server programs use the RTK Rail Abstraction Layer API. We compare the RTK version with a socket-based program. The socket program does not use the RTK API and relies on the default system affinity and scheduling heuristics.

1) Effects of Various Rail Configurations on Message Latency

Figure 6 compares the performance of setting the rails affinities on the message transfer latency for various payloads using four concurrent TCP (two per NIC). We compare this performance with the default case in Linux. As seen from the graph, as the payload size increase, thread and memory affinities help in reducing the transfer latency. For a payload of 8MB, thread and memory affinity together yield reduce the message latency by 33% over the default Linux case. This is mainly due to the improved cache locality and lower data access latency due to memory affinity. This is reduction in message latency is critical for applications using MPI and network-intensive applications. A similar trend is seen in the case of UDP in Figure 7 where thread and memory affinity together help reduce the transfer latency. The effect of thread affinity on latency is clearly visible as the payload size exceeds 4K (page size). The effects of memory affinity are not very pronounced in comparison to thread affinity due to the fact that the payload fits into the processor cache.

2) Effects of Various Rail Configurations on Throughput and CPU Utilization

The effect of affinities of the throughput of network-intensive TCP and UDP workloads is shown in Figures 8 and 9. The workload consists of four concurrent streams (two per NIC). We compare the achievable throughput for the stream using the RTK Abstraction Layer API to enable affinities (at both the sender and the receiver) with the achievable throughput on the system relying on default system settings. We notice that enabling affinities improves the achievable

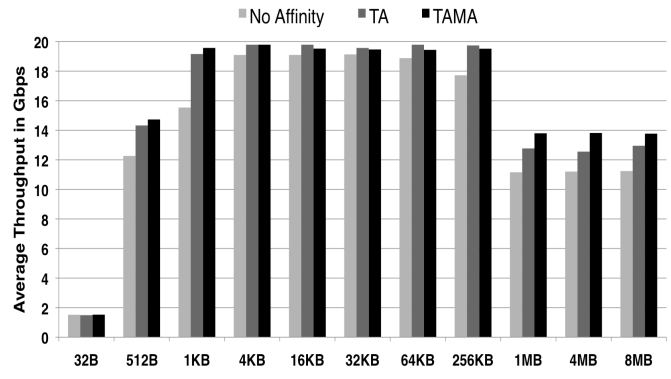

Figure 8. Effect of affinities on the aggregate throughput of four TCP streams. As the payload size increases, enabling affinities leads to higher throughput in comparison to a default Linux system.

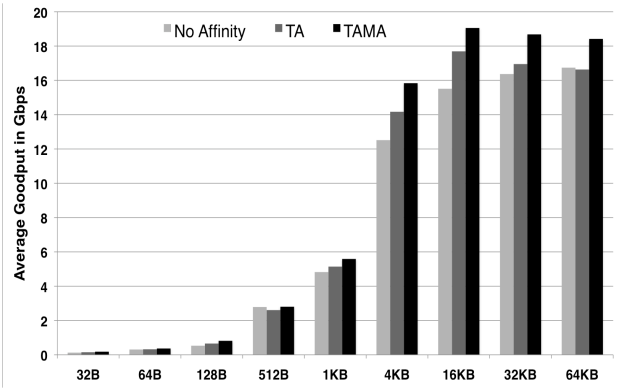

Figure 9. Effect of affinities on the aggregate goodput (thoughput with 0% packet loss) of four concurrent UDP streams. As payload increases, enabling affinities leads to a higher performance in comparison to a default Linux system.

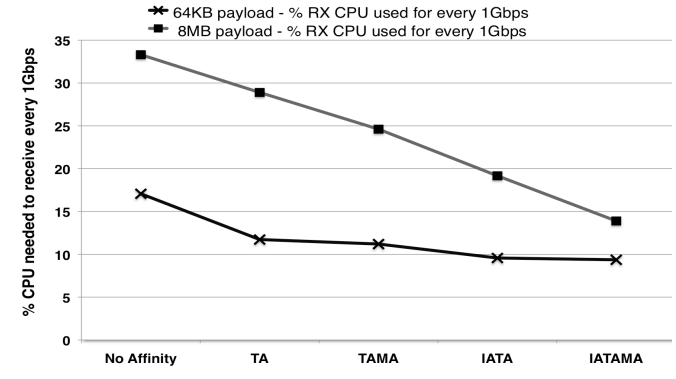

Figure 10. Effect of affinities on the average CPU Utilization of four concurrent TCP Streams to process 1Gbps. Using RTK, the application needs less CPU to process 1Gbps TCP traffic.

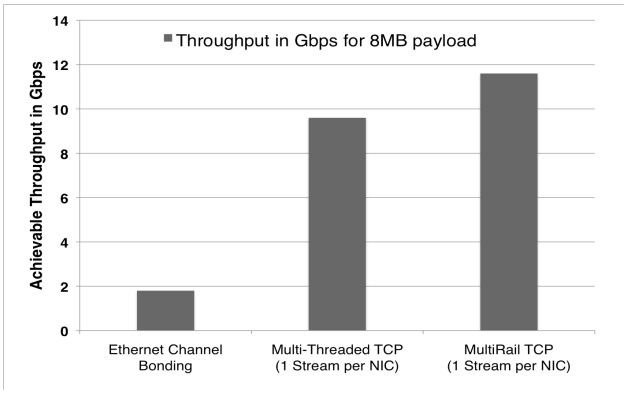

Figure 11. Performance evaluation of 2 x 10GE Linux Ethernet Channel Bonding, user-space Multi-Threaded TCP over 2 NICs and user-space MultiRail Library based TCP over two network rails for transferring a payload of 8MB.

throughput. This improvement is significant for higher payloads. In case of TCP, for a payload of 256 KB, affinities result in an improvement of 2 Gbps over the default settings. This is primarily due to factors, including lower cache pollution and lower memory access latency. Thus, allocating resources taking the topology into account is critical for network intensive e-Science and throughput intensive applications including wide-area data transfer.

Figure 10 depicts the effect of affinities on the average CPU utilization of the network streaming at the receiver to process 1 Gbps. In this experiment, we have 4 concurrent streams competing to process the network streams. This is very common in cluster-based applications. Lower CPU usage by the network application would yield precious CPU cycles for the compute-intensive components. Enabling affinities results in a reduced CPU usage, which is primarily due to reduced contention of resources and lower cache pollution. In case of 8MB payload, a fully affine system leads to the 50% reduced CPU usage. This is mainly due to the fact that memory affinity leads to low-latency data access. This is critical for cluster-based applications wherein precious additional CPU cycles (leveraged from an affine network component) can be dedicated to compute-intensive components.

3) MultiRail TCP benchmarks

With the advent of Multi-lane NICs, efficient methods to exploit parallelism throughout the system and networks are necessary. This is critical for future LambdaGrids based applications using IP over Ethernet over DWDM. The current options include, Ethernet channel bonding and designing a multi-threaded protocol. MultiRail-TCP is a simple socket-like API that allows applications to leverage the performance benefits of rails without having to significantly modify an application's source code. Internally, it takes advantage of data parallelism by splitting the data onto multiple streams; task parallelism by creating worker threads to stream the data; and network parallelism by using the multiple NICs available on the system. Similar to the micro-benchmarks, our MultiRail-TCP experiment created network rails with thread and memory affinity. The memory was interleaved between the two memory banks. As seen in Figure 11, MultiRail-TCP achieves a throughput 2 Gbps higher than a Multi-Threaded TCP over 2 NICs. This is due to the fact that the network rails reduce resource contention that is present in Multi-Threaded TCP wherein the threads are scheduled based on the Linux scheduler's heuristics. MultiRail-TCP achieves 10 Gbps higher throughput than Linux Ethernet channel bonding. This is primarily due to the locking overheads in the Linux channel-bonding driver. Linux channel bonding works at gigabit rates but fails to scale to 10Gbps rates. Thus, one can achieve high performance at a user-space by exploiting parallelism throughout the system.

B. Application Benchmarks – GPU Data Streaming

Accelerators including GPUs are increasingly becoming prevalent in e-Science. GPUs are used for computation, information visualization and collaboration between scientists. Efficient data streaming between GPUs on multiple nodes, and, streaming between the GPU and the CPU is critical in a GPU's performance in HPC clusters. We evaluate the efficacy of the RTK API on the performance of the "Netvideo" GPU streaming application. Netvideo is used for streaming 4K frames (4096 by 2048 pixels) of supercomputing e-Science simulations for remote and interactive visualization by scientists. This is very useful for steering simulations, especially in the petascale era.

4K visualization at 24 frames per second (fps) in RGBA format (32bit per pixel, with red, green, blue and alpha channels at 8-bit each) requires 6.4 Gbps of network bandwidth to stream from the rendering site to the display site. 'Netvideo', shown in Figure 12, consists of:

- A sending application, streaming 4K frames from main memory (usually simulation data).
- A receiving application that receives the frames, and downloads them to the graphics card for display. To optimize the pixel download, Netvideo uses pixel buffer objects (an OpenGL feature) that offers asynchronous DMA transfer to the GPU through its PCIe link.
- A 10G Myrinet interconnect between the sending and receiving machines for data transfer.

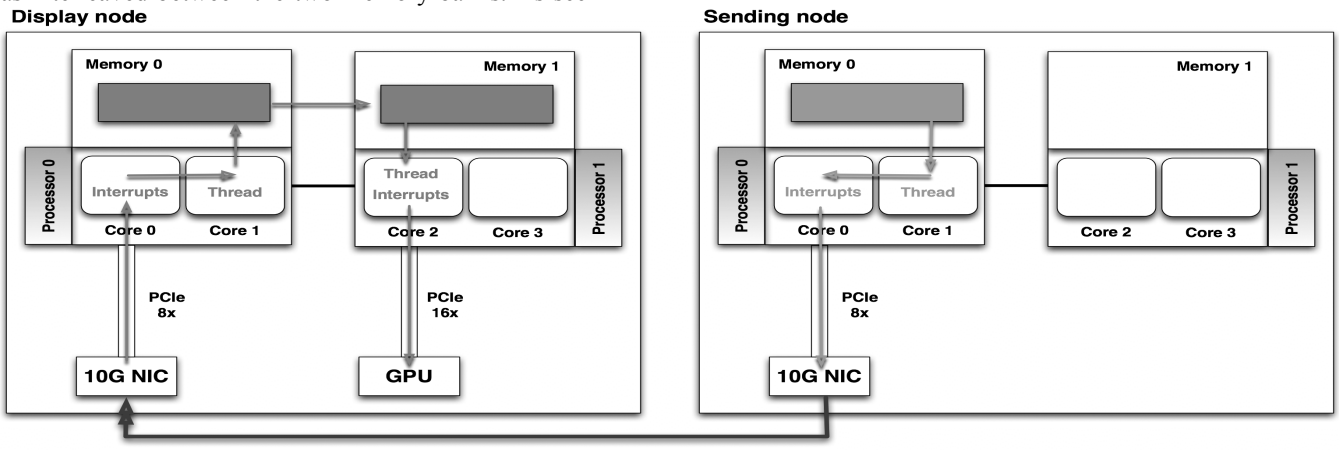

Figure 12. Streaming data from a GPU to a remote node over 10Gbps network using the RTK API for efficient resource co-scheduling. RTK improves the performance of GPU data streaming by 11% over default system scheduling by co-scheduling GPU, memory and network subsystems

In this scenario with multiple devices requiring very high throughput, namely the network interface and graphics card, the rails approach helps in reducing the contention between the various devices resulting in an improved performance.

The results are as following:

- The default Linux mechanism balances the interrupt load between devices (IRQ balance daemon) without any affinities and yields an end-to-end (from memory to remote display) bandwidth of 5.6Gbps (approximately 21 fps).
- Using RTK API, Netvideo achieves an end-to-end throughput of 6.2 Gbps (approximately 23 fps). This is close to interactive data visualization.

IV. Related Work

Libnuma API [6,7] is a user-space library for NUMA memory allocation in Linux. It supports two levels of memory hierarchy, namely local memory and remote memory. Thus, Libnuma fails to abstract multi-level memory hierarchy, including the three-levels of memory hierarchy in quad-processor systems [7,8]. Additionally, we found a bug in the default allocation of Libnuma. Instead of allocating memory on the node where the thread was last scheduled, Libnuma always allocates memory onto Node 0 by default. In comparison, RTK supports multi-level memory hierarchy and also abstracts hybrid combinations of NUMA and UMA characteristic of future architectures. Message Placement Optimization (MPO) [9] is an implementation of NUMA in Solaris. MPO abstracts the multi-level memory hierarchy available and exposes this to user applications. In addition to abstracting the memory hierarchy, the Rails toolkit enables co-allocation of buffers with other sub-system resources including threads and interrupts. It also provides a memory buffer class to user applications, which facilitates seamless run-time modification to a buffer's policies. This is essential for adaptive scheduling algorithms.

PLPA in OpenRTE [10] presents an API for binding threads to processors. However, OpenRTE has been discontinued as a standalone project and is tightly integrated into OpenMPI. RTK's thread binding has been influenced by OpenRTE's design and presents a similar intuitive API as a C++ object.

Multirail networks using technologies, including, Infiniband, Quadrics and Myrinet, have been used in HPC clusters to overcome the bandwidth bottleneck [5, 11]. In this case, a rail refers to a single network path and the payload is striped over the multiple network paths between any source-destination pair. In our work in the MultiRail Network library, in addition to exploiting parallelism of the network interconnects, we also exploit task parallelism critical for performance in multi-core environments.

V. Conclusion

The Rails Toolkit (RTK) abstracts the system topology and presents applications with an intuitive API to efficiently exploit end-system topology. RTK provides interfaces at multiple levels for integration with applications and middleware. The RTK API results in improved throughput, lower CPU usage and reduced message latency for applications by efficiently co-allocating resources. These results can help guide the development of future intelligent middleware that will automatically optimize the performance based on system parameters and conditions, thus making it easier for applications developers, who are often not systems experts, to make full use of the system capabilities. We are currently working towards extending RTK to other system architectures, including on-core GPU and accelerators, to enable future e-Science applications achieve scalable performance.

Acknowledgments

We would like to thank Alan Verlo (EVL, UIC), Lance Long (EVL, UIC), Maxine Brown (EVL, UIC), Andrew Johnson (EVL, UIC), Patrick Hallihan (EVL, UIC), Laurin Herr (PII), Natalie Van Osdol (PII), Osamu Ishida (NTT), Kazuaki Obana (NTT), Kazuo Hagimoto (NTT), Larry Smarr (CalIT2, UCSD), Tom DeFanti (CalIT2, UCSD). This material is based upon work supported by the National Science Foundation (NSF), awards CNS-0420477, OCI-0441094, and OCI-0225642, as well as funding from the State of Illinois, Pacific Interface and Sharp Laboratories of America. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Modeling, Analysis and Prediction Project 2006, NASA Goddard Space Flight Center. http://map06.gsfc.nasa.gov

[2] B. Jeong, L. Renambot, R. Jagodic, R. Singh, J. Aguilera, A. Johnson, J. Leigh. "High-Performance Dynamic Graphics Streaming for Scalable Adaptive Graphics Environment," In the proceedings of IEEE/ACM Conference on Supercomputing 2006, Tampa, FL, November 11-17, 2006

[3] E. He, J. Leigh, O. Yu, and T. DeFanti, "Reliable Blast UDP: Predictable High Performance Bulk Data Transfer," In *Proc. of the IEEE Conference on Cluster Computing*, September 23 - 26, 2002.

[4] V. Vishwanath, J. Leigh, E. He, M.D. Brown, L. Long, L. Renambot, A. Verlo, X. Wang, T.A. DeFanti, "Wide-Area experiments with LambdaStream over dedicated high-bandwidth networks," In Proceedings of IEEE INFOCOM High-Speed Networking Workshop: The Terabit Challenge 2006, Barcelona, Spain, April 24-26, 2006.

[5] J. Liu, A. Vishnu and D. K. Panda, "Building Multi-rail InfiniBand Clusters: MPI Level Design and Performance Evaluation," ," In the proceedings of IEEE/ACM conference on Supercomputing 2004.

[6] Linux NUMA policy. Linux manual pages.

[7] A. Kleen, "An NUMA API for Linux," 2004.

[8] U. Drepper, "What Every Programmer Should Know About Memory," 2007.

[9] J. Chew, "Message Placement Optimization - Solaris performance," http://opensolaris.org/os/community/performance/mpo_overview.pdf

[10] R. Castain, T. Woodall, D. Daniel, J. Squyres, B. Barrett, and G. Fagg, "The Open Run-Time Environment (OpenRTE): A transparent multicluster environment for high-performance computing," Future Gener. Comput. Syst. 24, 2 (Feb. 2008), 153-157.

[11] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits, "Using Multirail Networks in High-Performance Clusters," In Concurrency and Computation: Practice and Experience, 15 (7-8): 625--651, 2003