Throughput Optimization with a NUMA-Aware Runtime System for Efficient Scientific Data Streaming

Hasibul Jamil
University at Buffalo (SUNY)
New York, USA
mdhasibu@buffalo.edu

Joaquin Chung
Argonne National Laboratory
Illinois, USA
chungmiranda@anl.gov

Tekin Bicer
Argonne National Laboratory
Illinois, USA
tbicer@anl.gov

Tevfik Kosar
University at Buffalo (SUNY)
New York, USA
tkosar@buffalo.edu

Rajkumar Kettimuthu
Argonne National Laboratory
Illinois, USA
kettimut@anl.gov

ABSTRACT

With the surge in data generation rates from advanced scientific instruments, there is an urgent need for effective network management and resource utilization strategies for data streaming. Present strategies often lag behind hardware advancements, leading to resource underutilization. Modern servers typically employ non-uniform memory access (NUMA) multiprocessors, which, despite their benefits, can pose performance challenges. This paper presents a novel runtime system tailored for efficient multi-stream data management, optimizing both its compression and decompression phases, and enhancing network I/O based on the server's unique hardware design. Our system coordinates parallel tasks for data compression, decompression, and transfer, aiming to reduce network data influx. Empirical tests show that aligning streaming tasks with the right NUMA domain results in a 1.48X throughput boost compared to cutting-edge methods and a 2.6X improvement over standard techniques.

CCS CONCEPTS

• **Hardware** → **Networking hardware**; • **Computer systems organization** → **Multicore architectures**;

KEYWORDS

Heterogeneous architectures, data compression/decompression, data streaming, runtime systems, performance optimization, non-uniform memory access (NUMA).

ACM Reference format:
Hasibul Jamil, Joaquin Chung, Tekin Bicer, Tevfik Kosar, and Rajkumar Kettimuthu. 2025. Throughput Optimization with a NUMA-Aware Runtime System for Efficient Scientific Data Streaming. In *Proceedings of ACM/IEEE Conference, Denver, Colorado, USA, November 2023 (SC2023 (INDIS workshop))*, 11 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large-scale scientific instruments, such as the detectors at the Advanced Photon Source (APS) in Argonne National Laboratory (ANL), are now generating vast volumes of experimental data at an unprecedented pace, often exceeding rates of 25 Gbps per beamline[14, 23]. This trend is projected to grow exponentially with the emergence of next-generation synchrotron radiation facilities, like the upcoming upgrade to the APS [41, 42]. The enhanced x-ray brightness, anticipated to be 500 times greater, is expected to drive corresponding increases in data-intensive imaging experiments. This may potentially lead to terabit-scale data generation rates and petabyte-scale experimental datasets [2, 24, 26].

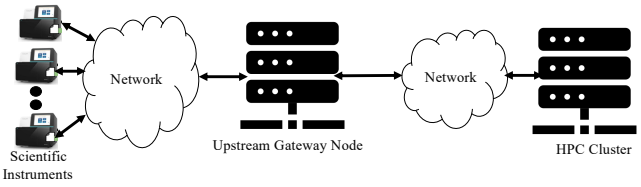


Figure 1: Multiple detectors at the Advanced Photon Source (APS) are streaming data to an upstream gateway, where data is accumulated for pre-processing or load-balancing before being forwarded to an HPC cluster. Within the HPC cluster, the data undergoes further analysis and processing, transforming the raw information into valuable insights.

The pace at which scientific instruments generate data increases faster than the capacity of the links and processing resources between the instruments and high-performance computing (HPC) resources. This creates a bottleneck in handling large datasets that can generate unwanted interruptions during data acquisition or force a manual reduction of the acquisition rate. Figure 1 illustrates a data streaming pipeline with an upstream gateway node. Positioned between the data streaming sources and the HPC cluster, this gateway node offers functionalities such as data aggregation, pre-processing, and load balancing, as well as serving as a security barrier. One promising approach to overcome these challenges involves optimizing the upstream system's architecture and effectively utilizing network bandwidth. By incorporating non-uniform memory access (NUMA) and integrating high-speed or multiple Network Interface Cards (NICs), the system can effectively increase the number of available cores and memory and available network bandwidth from a single host perspective. This, in turn, enhances the system's overall capacity to process and manage the influx of data, positioning it better to handle the ever-growing demands of modern scientific experimentation.

Traditional approaches for optimizing data streaming, which encompass tasks such as compression, decompression, and network I/O, typically depend on the operating system to assign specific

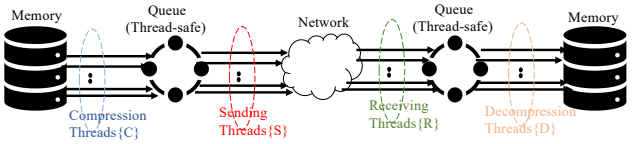


Figure 2: Schematic of the runtime system handling data streaming and processing tasks. The runtime system is formulated as a heterogeneous software pipeline. Uncompressed scientific data is streamed into memory, where a set of compression threads ({C}) compresses specific data chunks. These compressed data chunks are enqueued in a thread-safe queue, ready for a set of sending threads ({S}) to transmit them over the TCP/IP network to the upstream node. Upon arrival at the upstream node, a set of receiving threads ({R}) retrieves the chunks and places them into another thread-safe queue. A set of decompression threads ({D}) then decompresses each data chunk and stores it back into memory or disk.

cores for the execution of these tasks [10, 12, 13]. They often employ non-NUMA-aware memory allocation and follow static data paths. Originally crafted for uniform hardware setups, these methods fall short in addressing the complexities of contemporary heterogeneous systems. The rise of advanced NICs and the nuances of NUMA architectures have underscored the limitations of such rigid strategies. They often result in task-to-resource mismatches, leading to resource underutilization, inefficient memory access, and compromised performance [8]. In modern systems that incorporate NUMA, memory latency is inherently affected by the spatial relationship between the cores that request data and the target memory controllers [18, 25, 37]. In this context, maintaining local memory access becomes critical. Such an approach is not merely about minimizing latency; it also involves distributing data evenly amongst memory controllers. This balance helps in avoiding potential inter-socket contention, a situation where different parts of the system compete for the same resources [16, 39]. The implications of this are significant: to optimize memory access, NUMA-aware memory allocation needs to be closely aligned with resource-aware task-to-core mapping. Only by merging these two strategies, we can create a system that operates at peak efficiency. Through this harmonious integration, it is possible to enhance not only the overall performance of a system but also the utilization of its resources, leading to a more effective streaming operation.

With high-bandwidth single nodes capable of both transmitting and receiving substantial amounts of data, the real challenge emerges in maximizing resource efficiency. To address this, we devise a runtime system tailored to manage data streaming applications, acting as an intermediary between the software program and the underlying hardware. The system, working in conjunction with the OS, oversees intermediary memory allocation schemes and task-to-NUMA-domain mapping, optimizing hardware resource utilization as well as network bandwidth utilization. Our runtime system is versatile enough to integrate computational tasks as well. For example, cores that are not needed for network I/O can be repurposed for computation operations such as data compression and decompression. This enhances not only the effective data streaming rate but also the overall utilization of resources. Consider a system operating at 100 Gbps; if some cores are employed for compression at a 2X compression ratio, the effective data transfer rate is effectively doubled to 200 Gbps. The seamless integration of compression tasks leads to a substantial reduction in the size of data chunks being streamed, thereby decreasing ingress traffic volume. This traffic minimization optimizes network resource utilization, facilitating the coexistence of multiple services on a single network, either provisioned or ad-hoc. Figure 2 illustrates the various components of our runtime system and provides a schematic representation of the intended operation.

Designing this runtime system in terms of different tasks such as compression, decompression, and network I/O raises two pivotal questions: (1) How can the organization of data streaming tasks be optimized relative to execution core selection so as to maximize network bandwidth utilization? (2) How can computation jobs such as compression and decompression be effectively integrated into the streaming task, enabling concurrent use of CPU resources and effectively utilizing network bandwidth to reduce network traffic?

To address these concerns, we emphasize our contributions in following areas:

- We introduce several observations in § 3.1, 3.2, 3.3, 3.4 to devise a scheme to optimize the organization of data streaming tasks. By strategically selecting execution cores, we aim to maximize network bandwidth utilization, thus directly addressing the first posed question.
- We embed computational operations such as compression and decompression within the data streaming process. This not only promotes the efficient use of CPU resources but also maximizes the utilization of network bandwidth, offering a solution to the second question.
- Our empirical analysis reveals that strategically selecting the number of streaming tasks and aligning them to the most suitable NUMA domain can increase the average throughput by 1.48X compared to state-of-the-art methods and 2.6X over traditional baseline techniques.

The remainder of this paper is structured as follows: Section 2 provides the background. In Section 3, we delve into the system's architecture and our design approach. Section 4 evaluates the performance of our runtime system. We explore other relevant studies in Section 5, and Section 6 concludes the paper.

2 BACKGROUND

In this section, we first explore the background of the NUMA architecture and examine the operation of NICs within this framework. Our goal is to highlight the implications of NUMA-specific network I/O performance.

2.1 NUMA Architecture

Modern servers are equipped with a NUMA architecture, wherein multiple CPU cores are organized into distinct sockets. Each socket possesses its own designated memory, termed *local memory*, coupled with an advanced memory controller. This controller facilitates access to the memory across all other sockets. When one socket accesses the local memory of another socket, it is referred to as *remote memory*. Accessing remote memory is inherently slower than accessing the local memory. This is due to the necessity of transferring

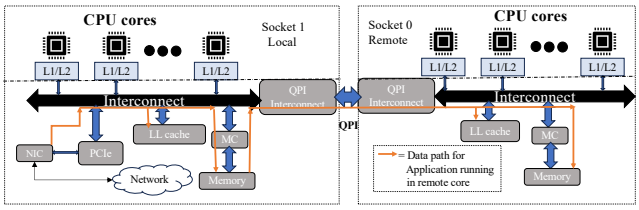


Figure 3: Network and memory I/O of the NUMA architecture. Throughout this paper, the terms 'NUMA socket', 'node', and 'processor' are used interchangeably to refer to the NUMA domain.

data from the remote memory through the NUMA interconnect. A bottleneck in cross-socket memory access arises when CPU cores in one socket access the memory of another socket via the Quick Path Interconnect (QPI) [1]. Each socket employs a memory controller (MC) to establish connections to its local memory channels, as depicted in Figure 3. Accessing the physical memory that is linked to a remote MC is called *remote memory access*. The QPI interfaces play a pivotal role in facilitating data transfers between sockets.

2.2 NIC Operation in NUMA Architecture

Once network packets reach end hosts, processing packets via NIC involves several stages [8]. Initially, packets are temporarily stored in the NIC's input buffer, typically an SRAM. Subsequently, the NIC retrieves an Rx descriptor from its queue, which indicates the virtual address where the packet should be transferred using Direct Memory Access (DMA) within the host memory. To perform this DMA, the NIC initiates PCIe write transactions using the packet descriptor's address. These transactions fall under the purview of the PCIe root complex, which remaps the virtual memory addresses to their physical counterparts with the assistance of an IOMMU. Once the physical memory address is deciphered, the root complex manages the transfer of the packet's data to the host memory. Upon successful transfer, a hardware interrupt activates an interrupt handler linked to a processor core. This handler then prepares a softIRQ context for its native core or an alternate CPU core. Every CPU core inspects its poll queue using a designated poll method and subsequently processes the queued softIRQ context.

It is worth noting that contemporary NICs utilize the multi-queue technique, supporting numerous receive and transmit descriptor queues. For each incoming packet, the NIC controller formulates a hash value. Using these hash values as a guide, the NIC ensures that packets from an identical data stream are directed to a specific queue while simultaneously distributing varied traffic flows evenly across multiple queues. Two key strategies, Receive-side Scaling (RSS) [5] and Receive Packet Steering (RPS), are employed to optimize network transmission performance in multi-core server systems. While RSS allows each NIC queue to be linked to a dedicated CPU core, RPS designates a specific core for managing a softIRQ context. Consequently, the receiving thread of the streaming application accesses this host memory to acquire the streaming data.

As shown in Figure 3, the NIC establishes a connection with the NUMA 1 domain. This implies that the host memory, where the root complex relocates the data, is situated within the NUMA 1 domain. As a result, receiving threads that are pinned to NUMA 1 cores can swiftly access the packets from their local memory. However, for those threads tethered to NUMA 0 cores, packet processing latency may increase due to the cross-socket phenomena or remote memory access dynamics.

3 SYSTEM ARCHITECTURE AND DESIGN SPACE EXPLORATION

Our runtime system is architecturally formulated as a heterogeneous software pipeline, integrating different tasks, including compression, transmission, reception, and decompression. As data moves through the system, individual data chunks go through these tasks in a pipelined fashion provided by our runtime framework. As shown in Figure 4, this architectural framework is not restricted to a single node. It is expansively distributed, stretching across various nodes, which include nodes explicitly dedicated to data generation (data streaming sender) and data reception (data streaming receiver). An integral part of our design is the 'runtime configuration generator,' which is responsible for generating the configurations for both the sender and receiver nodes. These configurations contain information related to the type of tasks designated to individual sockets, the number of tasks, and the task execution location. Figure 2 provides a breakdown of the individual components and tasks that constitute our runtime system. This includes the operations involved in compression, data sending, reception, and the final stage of decompression. These components, inherently modular, find their strategic placements either at the sender or the receiver of our data streaming runtime framework. The structural design and operational dynamics resemble with the paradigms of a heterogeneous software pipeline, as elaborated in [35].

The runtime system is implemented in the C programming language, tailored specifically for Linux operating systems. This ensures compatibility and efficient execution within the Linux environment. It is architected to be versatile and able to support a variety of workloads, notably those requiring compression and decompression, network tasks, and meticulous adjustments of each task's CPU affinity. For our networking operations, we utilize the zeroMQ [7] library, which provides a robust and high-performance messaging protocol. To optimize data transfer speeds, the lz4 [19] library is incorporated for swift data chunk compression and decompression. Additionally, we leverage the libnuma [4] library for precise control over CPU-thread affinity, enhancing the system's performance by taking advantage of non-uniform memory access (NUMA) architectures. `numa_bind()` has been used specifically to restrict task and its children to run and allocate memory exclusively from the specified NUMA sockets. Our implementation code is available at `https://github.com/H-jamil/ha4hpdt.git`.

In the subsequent section, we delve into the optimal configuration generation for both the sender and receiver ends of the data streaming pipeline. We examine various design spaces, considering differing tasks, task counts, and execution locations, along with their respective performance metrics. Our objective is to highlight the criticality of addressing NUMA-specific performance considerations for both computational tasks (such as compression and decompression) and data transfer tasks. Addressing these nuances

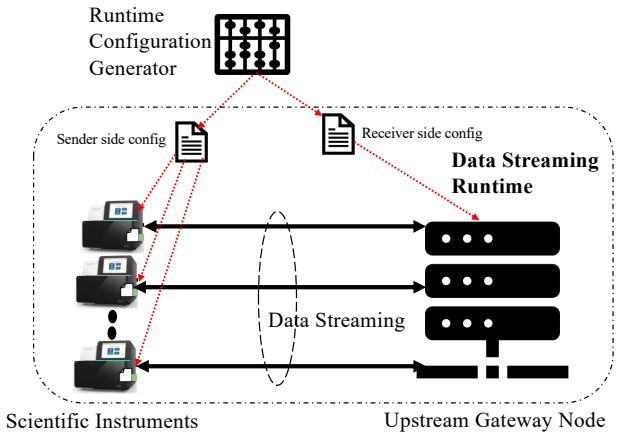


Figure 4: Architecture of the data streaming runtime system. The runtime system can span multiple nodes, with each node using a separate configuration file. These configuration files, generated by the runtime configuration generator, specify the task type, the number of tasks, and the location of task execution. Based on these configurations, both data producers and consumers initialize and conduct data streaming over TCP/IP network.

is pivotal for optimal resource utilization and meeting the performance demands of scientific data streaming.

3.1 Network performance and NUMA

We conduct experiments between APS and the Argonne Leadership Computing Facility (ALCF), two separate facilities within the Argonne National Laboratory (ANL). These facilities are interconnected by a network with 200 Gbps of bandwidth and 0.45ms RTT. Our aim is to investigate the effects of network transfer throughput and core affinity on the streaming process.

On the sending side, we employ four distinct machines to generate streams, simulating the data generation typical of scientific instruments and adequately matching the bandwidth capabilities of the receiver-side NIC. For processing source data, we employ the hdf5 [3] library, which allows for seamless management of large and complex datasets. Our receiving machine, which mirrors the role of the upstream machine in a scientific data streaming workflow (see Figure 1), is composed of two NUMA sockets. Each of these sockets has a Xeon Gold 6346 CPU with 16 physical cores operating at 3.1GHz (equivalent to 32 threads). Additionally, each socket boasts 512GB of memory (broken down as 16x32 DDR4 32GB 3.2GHz ECC RDIMM) per socket/CPU. Each NUMA socket is also directly tethered to a Network Interface Card (NIC) via a PCIe 4.0 link. The dual-port Mellanox ConnectX-6 (MT28908) NIC provides a bandwidth performance of up to 200 Gb/s per NIC, resulting in a combined bandwidth of 400 Gb/s for both NICs. However, the NIC in the NUMA 0 domain connects to a LUSTRE file system through a separate network, this connection is not used in our study. Hence, our attention is primarily centered on the NIC connected to the NUMA 1 domain as this particular machine works as the upstream gateway node.

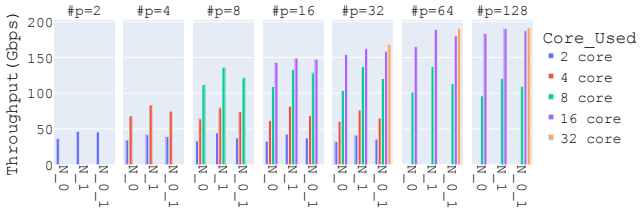


Figure 5: Depiction of the throughput achieved as the number of data streaming tasks varies across different NUMA domains. Here, #p indicates the number of streaming processes. 'N_0', 'N_1', and 'N_0_1' represent scenarios where all streaming processes are executed on NUMA 0, NUMA 1, and equally divided between NUMA 0 and NUMA 1, respectively. It is noteworthy that an average increase of 15% in throughput is observed when transfer tasks are allocated to cores in the NUMA 1 domain.

In a typical setup on the sender side, there are mechanisms (e.g., the NIC to CPU backpressure) in place to avert host congestion [16]. In this particular experiment, however, senders exclusively generate data chunks at a fixed rate, emulating data creation in scientific instruments. As a result, our primary interest lies in the receiver end of the streams, representing the upstream gateway machine as shown in Figure 1. In the conducted experiment between Polaris cluster (ALCF) and *lynxdtn* (upstream gateway node), a network throughput of over 190+Gbps is achieved on the receiver side. Throughout this experiment, we vary the number of processes and cores used to study their influence on achieved throughput. Each sending process has 1 sending thread, and each receiving process has 1 receiving thread. We increase from a base of 2 processes up to 128 processes across a range of 2 to all 32 available cores.

The data shown in Figure 5 reveals two main observations based on the number of processes that are running: (1) An increase in the number of streaming processes and utilized cores results in an increase in the receiver-side throughput. (2) Given the NIC's connection to the NUMA 1 domain, an average 15% boost in throughput is achieved when all streaming processes are assigned to cores within NUMA 1.

We examine both core utilization and the average remote memory access for each core during the streaming operation. These observations are illustrated in Figures 6 and 7. Due to space constraints, several configurations have been omitted. As anticipated and discussed in previous § 2.2, assigning streaming processes to cores in the NUMA 0 domain led to an overhead due to remote memory access, as shown in Figure 7. The NIC connects to the NUMA 1 domain, allowing quick packet access for threads on NUMA 1 cores but potentially increasing latency for threads on NUMA 0 cores due to cross-socket behavior. This overhead consequently resulted in a reduced throughput.

Observation 1: *Selecting the appropriate NUMA socket is crucial when deploying a data streaming application on a server. Streaming performance is influenced by the NUMA socket on which the receiving threads operate and the NUMA socket to which a particular NIC is connected.*

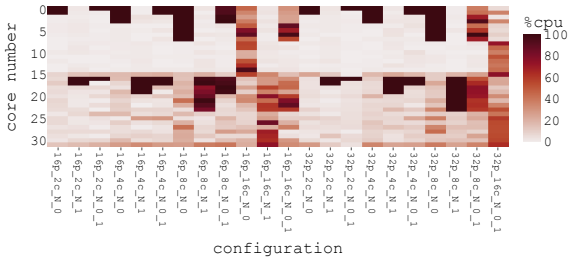


Figure 6: Core usage for different configurations during the experiments. The 32 cores are depicted in the Y-axis, with core 0 starting from the top. In the X-axis, the label 16*P*_2*c*_*N*_0 **denotes 16 streaming processes that are running in 2 cores from NUMA 0 domain.**

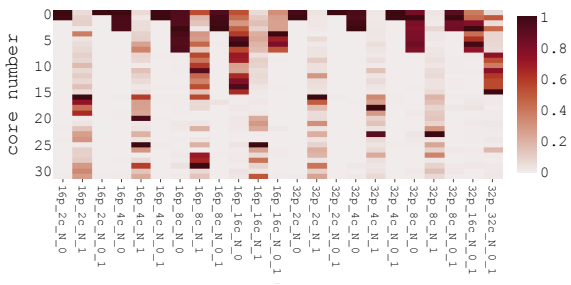


Figure 7: Average normalized remote memory access (i.e., NUMA access) bandwidth for every CPU core during different configurations of the experiments.

3.2 Compression performance and NUMA

The primary goal of the runtime system is to optimize the use of available resources and minimize network I/O. To achieve this, available CPU cores are employed to compress outgoing data and decompress incoming data chunks efficiently. This not only speeds up data movement but also makes better use of the compute resource utilization. For example, if a system can move data at a speed of 100 Gbps and unused cores work to compress the data to half its size, the result is that the effective data transfer speed is 200 Gbps. By shrinking data sizes, we reduce the amount of data being sent, making the network less congested. This helps multiple users and services to share the same network. In our experiment, we utilized a synthesized dataset of 16 GB, which mirrors real tomographic datasets outlined in [6, 36]. This data can be located either in the NUMA 0 memory domain or the NUMA 1 domain, serving as the source for our streaming operations. The data chunk size chosen for our streaming process is 11.0592 MB, equivalent to data from one X-ray projection. This chunk size represents the unit of operation in our streaming workflow. Compressor threads successively fetch sequential data chunks, which are then passed through the LZ4 algorithm [19] for compression. LZ4 is renowned for its speed, lossless compression, and favorable compression ratio. On average, the data stream achieves a compression ratio of 2:1 for the data chunks.

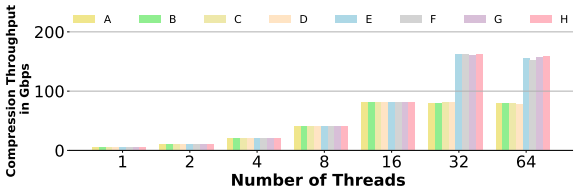
We examine the compression speed in relation to the number of threads CC, the memory location of the dataset, and the execution domain of the compression threads. The potential memory locations for the source data, as well as the chosen schemes for designating the compression threads to specific NUMA domains, are outlined in Table 1. This compression operation simulates the compression component of the sending machine within a runtime system tailored for a scientific instrument data-flow scenario. We conduct experiments for each configuration ten times and present the average results.

In the results shown in Figure 8, we look at how the compression throughput of data compression changes based on the number of threads {C} we use. We find that the more threads we use, the faster the data is compressed, but only up to a point. Specifically, once we use more threads than the number of available cores on the

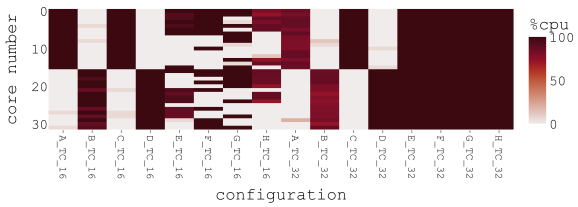
Table 1: Experimental configurations corresponding to Figures 8a, 8b, 9a, 9b. The 'Memory Domain' indicates the NUMA domain in which the data resides, while the 'Execution Domain' specifies the domain where the threads execute their operations. In configurations E and F, 'Execution Domain 0 & 1' indicates that threads are evenly distributed between NUMA domains 0 and 1. For configurations G and H, the operating system (OS) determines the thread execution domains.

Configuration	Memory Domain	Execution Domain
A	0	0
B	0	1
C	1	0
D	1	1
E	0	0 & 1
F	1	0 & 1
G	0	OS
H	1	OS

CPU, we do not see any more speed improvement. In the results depicted in Figure 8b, we observe the utilization patterns of all 32 cores under various configurations for 16 and 32 compression threads. Since both the NUMA 0 and NUMA 1 sockets possess 16 cores each, when the number of threads surpasses 16, all threads execute within the same domain, causing multiple threads to run on the same core. This leads to context switching, which explains the nearly halved performance for configurations A, B, C, and D when using 32 and 64 threads, compared to configurations E, F, G, and H, where threads can run concurrently across all 32 cores. Moreover, the location of data storage and the specific location of compression do not influence the compression speed. The uniform compression speed, irrespective of the data storage or specific compression location, can be attributed to data cache prefetching technology [27]. This technology optimizes performance by preemptively loading anticipated data into the cache, minimizing the latency typically associated with accessing data from varying storage points.



(a) Compression throughput achieved as the number of concurrent compression threads vary across different configuration as shown in Table 1.



(b) Core usage for different configuration during the experiments with thread number 16 and 32 for compression as shown in Figure 8a.

Figure 8: Relationship between compression threads {C}, as illustrated in Figure 2, and the achieved compression throughput. The compression throughput is directly proportional to the number of threads, provided the number of threads does not exceed the available CPU cores. The data residing domain and the domain of compression execution appear not to impact the compression throughput's performance.

Observation 2: *Data compression speeds up with increased threads only until the number of threads matches the CPU's core count; beyond that, performance declines due to context switching overhead. Additionally, source data storage location and compression execution location in the NUMA domain do not impact the compression performance.*

3.3 Decompression performance and NUMA

In our study, we also explore the decompression process of compressed data chunks. Our goal is to analyze how the decompression speed varied based on factors such as the number of decompression threads {D}, the storage location of the data chunks, and the processing location of the decompression threads. Details on possible memory locations for the compressed data and the strategies used to allocate decompression threads to specific NUMA domains are detailed in Table 1. This decompression process is representative of the actions performed by the receiving machine in a runtime system designed for data streaming in scientific instruments. For consistency, each configuration is tested ten times, and the average results are depicted in Figure 9. This Figure shows how the decompression speed of data chunks is influenced by the number of threads {D} employed. Our findings suggest that using more threads accelerates the decompression process, achieving a speed approximately 3X faster than the compression operation when using the same number of worker threads.

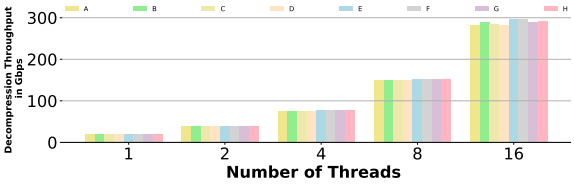
Further, Figure 9b highlights the utilization patterns of all 32 cores across diverse configurations, using 8 and 16 decompression threads. With 8 threads, performance remains consistent across the configurations detailed in Table 1. However, with 16 threads, configurations E and F outpace the others in throughput. The key differentiator in configurations E and F is the even distribution of decompression threads across both NUMA 0 and NUMA 1 sockets. This distribution minimizes intra-socket resource contention—specifically at the last level cache (LLC) and memory controller (which links domain memory to LLC)—compared to configurations like A, B, C, and D (where all decompression threads operate) or G and H (where the majority function within a single NUMA domain) [18]. This distinction is evident in Figure 9b. We capped our evaluation at 16 decompression threads, considering the decompression throughput achieved with 16 decompression threads is sufficient for the receiver end of our runtime system for handling individual streams.

Observation 3: *Decompression performance is unaffected by the NUMA domain location of the compressed (source) data chunk or where the decompression execution occurs. Increasing the number of decompression threads enhances throughput, especially when threads are evenly distributed across NUMA domains, minimizing resource contention.*

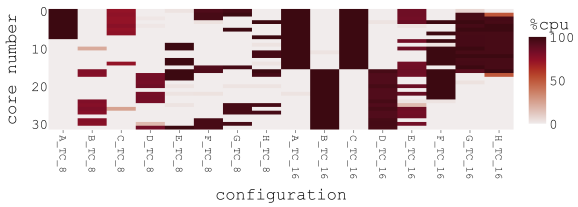
3.4 Sending and receiving threads and NUMA

We conduct additional experiments to understand how the number and execution location of sending and receiving threads influence network throughput in a streaming application using our runtime system. For this study, we focus solely on the sending and receiving operations, omitting the compression and decompression processes. This simulates the sender-side sending and receiver-side receiving operations in our runtime system, as depicted in Figure 10. The sending machine, $updraft1$, has a NIC supporting 100 Gbps, limiting our experiments to this maximum network bandwidth. The size of data chunks sent and received in this study equates to the average compressed chunk size. Furthermore, these chunks are stored in memory where the respective send and receive threads execute, based on Linux OS's first-touch policy. This policy dictates that a data page is allocated in the local memory of the core that first accesses it [30]. The potential execution locations for both sending and receiving threads across NUMA domains are outlined in Table 2. The number of sending and receiving threads is symmetrical; that is, for every x sending threads, there are x receiving threads, resulting in x TCP streams. Considering the shared nature of the network with other users and services, we repeat each configuration 30 times for consistency. The average results are showcased in Figure 11.

The relationship between the location of receiving threads and network I/O throughput reveals some interesting patterns. On the machine $updraft1$, the Network Interface Card (NIC) supports a maximum bandwidth of 100 Gbps. Consequently, as we increment the number of sending and receiving threads from 1 to 2, there is a sharp rise in transfer throughput across all configurations. However,



(a) Decompression throughput achieved as the number of concurrent compression threads vary across different configuration as shown in Table 1.



(b) Core usage for different configuration during the experiments for decompression with thread number 8 and 16 as shown in Figure 9a.

Figure 9: Relationship between decompression threads {D}, as depicted in Figure 2, and the achieved decompression throughput. The decompression throughput is directly proportional to the number of threads and on average ~3X greater than the throughput achieved with the same number of compression threads. Similar to the compression and NUMA scenario, the domain where the data resides and the domain where decompression execution takes place do not seem to influence the performance of the decompression throughput.

as the thread count increases from 2 to 3, not all configurations exhibit the same growth rate. Specifically, configurations B and D see a more subdued throughput increase compared to configurations A, C, and E.

A closer examination reveals that configurations B and D achieve a higher throughput when receiving threads are on NUMA domain 1, especially noticeable for thread counts of 1, 2, and 3. This behavior aligns with our earlier observation: having the NIC connected to NUMA domain 1 and executing the receiving threads in the same domain can boost throughput by up to 15%. Interestingly, the location where the sending threads execute does not influence the transfer throughput. This is attributed to specific mechanisms on the sender side, such as NIC to CPU backpressure, which effectively prevents host congestion [16]. In an alternate scenario where the NIC could have supported bandwidth exceeding 100 Gbps, we speculate that configurations B and D might have sustained their throughput growth rate even when employing three threads. However, as it stands, all configurations exhibit similar throughput patterns once the thread count hits 4.

Observation 4: *Network throughput is significantly influenced by the receiving thread's location, with configurations B and D seeing up to a 15% boost when threads operate within NUMA domain 1. Conversely, the sending threads' location has no discernible impact on throughput, likely due to sender-side mechanisms that mitigate host congestion.*

4 PERFORMANCE EVALUATION

From the insights obtained in the previous section, we generate configurations for different components of our runtime system using the runtime configuration generator. Our runtime system is a heterogeneous software pipeline, integrating tasks such as compression, sending, receiving, and decompression. Data chunks traverse multiple software components within our framework, extending across nodes, including data generation and reception, linked by a network. In this section, we first present the performance of our runtime system using a baseline configuration for a single-stream application. Then, we demonstrate its capability in handling multi-stream operations and compare its performance enhancements with state-of-the-art OS-based solutions.

Table 2: Experimental configurations corresponding to Figure 10. The 'Sender Socket' column indicates the NUMA domain in which the the sender threads are residing, while the 'Receiver Socket' column specifies the domain where the receiver threads execute their operations. In configurations E OS determines the thread execution domains both in sender and receiver side.

Configuration	Sender Socket	Receiver Socket
A	0	0
B	0	1
C	1	0
D	1	1
E	OS	OS

4.1 End-to-End Performance for a Single Stream

We assess the efficiency of our runtime system using a single data stream under varying configurations (e.g., differing numbers of compression and decompression threads, transmission-reception threads, and the domain of execution for the receiver thread) as outlined in Table 3. Our runtime system is integrated within two machines: $updraft1$, responsible for generating data, and *lynxdtn*, designed as the data consumer. $updraft1$ has same configuration and organization as *lynxdtn* depicted in §3.1. These machines are interconnected via a network that can accommodate 100 Gbps, as depicted in Figure 10. Each experiment was iterated five times, and the average result is presented in Figure 12.

In Figure 12, the X-axis represents the number of sending-receiving threads, and the color of each bar signifies the execution NUMA domain of the receiver threads. Notably, the observed trends reveal that, for configurations A and B, the achieved throughput remains relatively constant. This finding underscores the fact that maintaining a constant number of compression threads while increasing the count of decompression threads does not result in a notable increase in the end-to-end throughput. Moreover, it is evident that increasing the count of sending-receiving threads does not significantly impact the overall performance, implying that the bottleneck

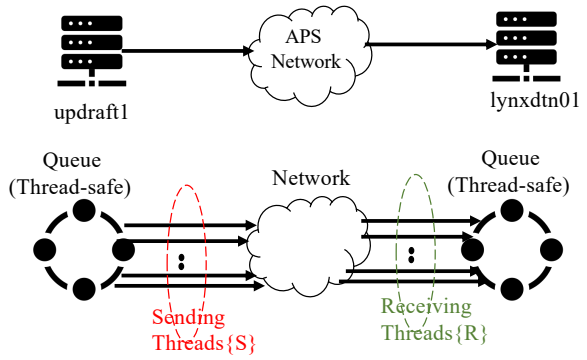


Figure 10: Experimental setup illustrating the relationship between the number of sender(*updraft*1) and receiver(*lynxdtn*) threads.

Table 3: Experimental configurations corresponding to Figure 12.

Configuration	#of compression Threads	#of decompression Threads
A	8	4
B	8	8
C	16	8
D	16	16
E	32	4
F	32	8
G	32	16

primarily arises from the number of compression threads for these configurations. As the number of compression threads is increased, the bottlenecks within the end-to-end pipeline shift across different segments. This phenomenon becomes particularly evident when analyzing configurations C and D. In this case, the throughput exhibits higher values when the receiver threads are executed within the NUMA 1 domain.

Furthermore, when the number of compression threads is constrained to match the available CPU cores on the sending machine, as is the case with configurations E, F, and G (e.g., 32 compression threads), the bottleneck dynamics change once again. The limitations are now imposed by the quantities of sending-receiving threads, decompression threads, and the execution domain of the receiving threads. Notably, in configurations F and G, with 8 sending-receiving threads and executing receiver threads within the NUMA 1 domain, we manage to achieve an end-to-end performance of 97 Gbps. This notable enhancement is 2.6X greater than the baseline performance achieved with configurations A and B, which yielded 37 Gbps.

4.2 Performance Comparison

To assess the effectiveness of our runtime system, we conduct a comparative analysis with a scenario where we refrain from explicitly designating the execution locations for runtime system

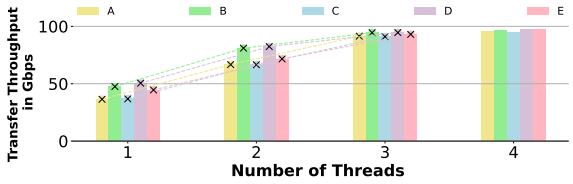


Figure 11: Network Throughput between two machines in APS network.

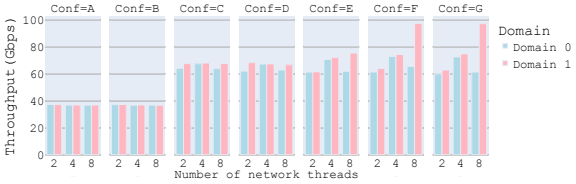


Figure 12: End-to-end throughput based on various configurations of the compression and decompression threads, as outlined in Table 3. The throughput is notably higher when the receiving threads are located in the NUMA 1 domain, as indicated by the red bar.

component threads (such as compression, sending, receiving, and decompression). Instead, we allow the operating system (OS) to determine the execution locations autonomously. In this context, our runtime system operates within the framework of five distinct machines: *updraft*1, *updraft*2, *polaris*1, *polaris*2, all engaged in generating four distinct data streams, while *lynxdtn* serves as the data consumer, assuming the role of the upstream gateway node.

Both *updraft*1 and *updraft*2 are characterized by similar configurations and organizational structures, mirroring those of *lynxdtn* as detailed in § 3.1. Conversely, the *polaris*1 and *polaris*2 nodes share a similar architecture and organization, each equipped with a 2.8 GHz AMD EPYC Milan 7543P CPU boasting 32 cores with 512 GB of DDR4 RAM. All four sender nodes have NICs with 100 Gbps. However, it is noteworthy that only the *lynxdtn* machine as a receiver has a NIC of 200 Gbps. *updraft*1, *updraft*2, and *lynxdtn* operate on Red Hat Enterprise Linux 8, employing kernel version 4.18. Conversely, the *polaris*1 and *polaris*2 machines opt for SUSE Linux Enterprise Server 15 SP3, with kernel version 5.3.

The interconnection among these machines is realized through a real network path capable of accommodating a bandwidth of 200 Gbps, as illustrated in Figure 13. This comprehensive setup allows us to holistically evaluate the performance of our runtime system and draw meaningful comparisons in terms of network utilization and end-to-end performance, as shown in Figure 14.

In Figure 14, we present a bar graph depicting the combined and individual network throughput, along with the end-to-end throughput for each of the four streams: stream-1, 2, 3, and 4. With an average compression ratio of 2:1, it's noteworthy that after decompression, the end-to-end throughput becomes twice that of the network throughput.

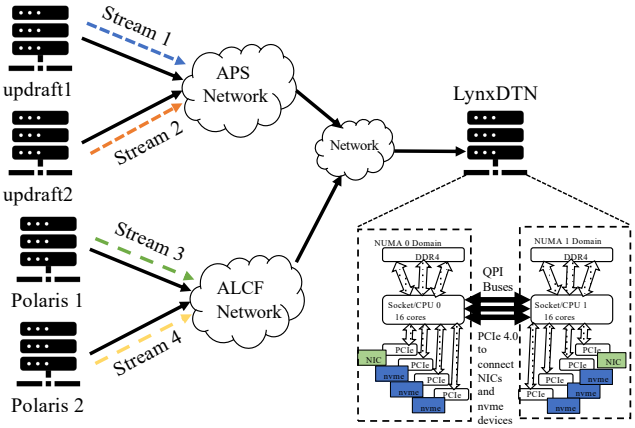


Figure 13: Experimental setup illustrating the generation of four concurrent data streams using four distinct machines. One machine, denoted as *lynxdtn*, acts as the upstream gateway, receiving all the overlapping streams.

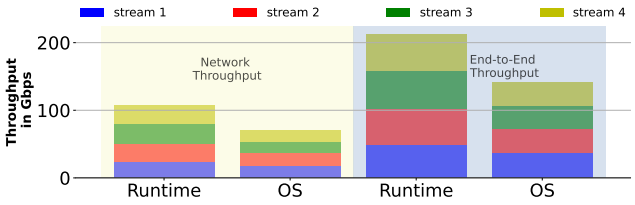


Figure 14: Data streaming performance with the setup specified in Figure 13. Evaluation of network performance and end-to-end performance based on receiving threads and decompressing threads execution location chosen by OS versus specified by our runtime system. In all cases, the sender uses 32 compression threads and 4 sending threads.

Given that the NUMA 1 domain of the *lynxdtn* machine is equipped with 16 cores, and considering the presence of four distinct data streams, we have allocated these cores evenly among the streams. This approach justifies our choice of utilizing four sending-receiving threads. This decision is grounded in the notion that running multiple sending-receiving threads on a single core introduces context-switching overhead, ultimately detrimentally affecting network I/O performance, as elaborated in §3.1. Each stream, correspondingly, employs four decompression threads assigned to four cores residing in the NUMA 0 domain, so all 16 cores in NUMA 0 domain are evenly distributed between four data streams.

In the case of the OS, we specify the number of threads, and the OS determines the execution locations for individual threads. Specifically, we found that the operating system, while capable of determining thread execution locations, does not always possess the intricate architectural and organizational knowledge of the involved machines to maximize efficiency. Our runtime system, on the other hand, is designed to tap into such knowledge, enabling it to manage not only the optimal number of threads (compression, sending, receiving, decompression) but also their precise placement within the most suitable NUMA domains. This approach yields significant performance improvement, as shown in Figure 14. Specifically, we achieve 105.41 Gbps cumulative network performance, corresponding to 212.95Gbps cumulative end-to-end performance. In contrast, relying solely on the OS, the network throughput reaches 70.98Gbps, accompanied by an end-to-end performance of 143.3Gbps. Evidently, our runtime system outperforms the OS-based approach by a 1.48X factor. This performance improvement is due to runtime system's capacity to harness knowledge regarding the architectural and organizational knowledge of sender and receiver machines. This enables the runtime system to expertly manage the execution of the optimal number of compression, sending, receiving, and decompression threads, in addition to strategically placing them within the most suitable NUMA domains.

5 RELATED WORK

As high-speed data sources proliferate, a range of data-intensive applications are being implemented in real-world scenarios. These applications, with their stringent latency and throughput demands, cannot be adequately supported by traditional batch processing models. Notable efforts to enhance Data Stream Processing Systems (DSPSs) have been made by both the research community [29, 43] and industry giants like SAP[45], Google [17], and Microsoft [9]. However, the ever-increasing performance needs, complex analyses, and heavy state access requirements of emerging stream applications [11, 20, 32, 38] present new challenges. Although strides in computer architecture have sparked a wave of interest in hardware-conscious DSPSs [43, 44], aiming to exploit modern hardware capabilities for stream processing acceleration, the heterogeneity introduced by the presence of NUMA and NUMA-to-NIC connection domain is often overlooked.

Previous research has explored the concept of NUMA-aware system optimizations within the field of relational databases [21, 31, 34]. RING [33], a NUMA-aware Message-batching runtime system, is designed to improve efficiency, primarily for irregular applications, by managing memory through a partitioned global address space model and leveraging one-sided RDMA API. [15] shows how coherence traffic can be constrained in a large, real cache-coherent NUMA (ccNUMA) platform comprising 288 cores by utilizing a combined hardware/software approach. [8] provides evidence of host congestion in production clusters, attributing it to the adoption of high-bandwidth access links, which leads to bottlenecks within the host interconnect (NIC-to-CPU data path). In contrast, our proposed runtime system introduces a specialized approach. It holistically integrates computational and I/O operations, emphasizing efficient multi-stream data transfers between scientific instruments and HPC clusters, and strategically orchestrates parallel tasks, significantly reducing the data volume on the network.

[28] describes a NUMA-aware thread and resource scheduling methodology to optimize data transfers over terabit networks. [40] presents that binding processes to the local processor, rather than to specific cores, improves the efficiency of high-speed data transfers. [22] delves deeper to understand the end-system bottlenecks for

high-speed TCP flows, emphasizing the significant role of affinitization, or core binding, on protocol processing efficiency and how it changes the performance bottleneck of the network receive process.

In the Network Functions Virtualization (NFV) domain, [18] discusses the challenges posed by the NUMA architecture in multi-core servers for service function chain (SFC) placement. [37] highlights the importance of considering NUMA architecture when deploying network processing software for NFV. It introduces the use of DPDK to enhance data plane performance and addresses the need for proper thread mapping on physical cores across NUMA sockets. [25] points out the challenges faced when deploying NFV on modern NUMA-based Standard High Volume Servers (SHVS), and proposes a collaborative thread scheduling mechanism to minimize end-to-end performance slowdown for NFV traffic flows, demonstrating improved CPU utilization and traffic throughput.

The above-mentioned studies provide valuable techniques and execution models but do not directly address our specific problem - managing the heterogeneity brought about by NUMA and NUMA-to-NIC connection domain for scientific data streaming applications. In contrast, our proposed runtime system operates as a layer above the operating system, specifically designed to accommodate this heterogeneity. It maintains a knowledge base of the underlying hardware, including NUMA configurations and NUMA-to-NIC connection domain, and can accordingly adapt data streaming and computational resource allocation. This enables the support of tasks like data compression, maximizing bandwidth utilization, and significantly improving the overall efficiency of data streaming tasks.

6 CONCLUSION AND LOOKING FORWARD

In an era marked by unprecedented data generation rates, especially from sophisticated scientific tools, the challenge of optimizing network and resource management for data streaming has never been more critical. Conventional models are lagging, often overwhelmed by the pace of advancements in hardware technologies. This lag can be seen in the common underutilization of resources and the consequent suboptimal system performance. Despite their inherent benefits, NUMA systems present challenges, especially when accessing memory segments located remotely from the processing unit. Addressing this, our research presents a runtime system capable of managing efficient multi-stream data management, ensuring optimal compression, decompression, and streamlining based on the specific hardware characteristics of the host server. The core of our suggested runtime system consists of data compression, decompression, and transfer tasks. Importantly, these tasks aim to limit the amount of data coming into the network. We report two major findings that we address with our runtime system: (1) The strategic allocation of data streaming tasks, in line with the selected execution CPU, can significantly improve network bandwidth utilization. (2) Incorporating data compression and decompression into the streaming process optimizes both CPU resource consumption and network bandwidth efficiency. Our empirical evaluations validate these hypotheses, suggesting that the pinning streaming tasks with the right NUMA domain can boost throughput by 1.48X in comparison to state-of-the-art solutions and 2.6X over baseline configurations.

As we move into a more digital world, we need systems like ours that are flexible and work efficiently. Looking ahead, our future work will focus on developing the dynamic capabilities of our runtime system. We aim to enable the runtime system to adjust the allocation of cores to streaming software processes in response to real-time resource utilization. By closely monitoring the usage of CPU cores, our runtime system will be able to react to varying processing demands, further optimizing resource management and improving the overall performance of the data streaming processes. This dynamic adjustment will introduce a level of adaptability to the runtime system, allowing it to better serve in data-intensive environments where computational needs may fluctuate.

ACKNOWLEDGMENTS

This material was based upon work supported by the U.S. National Science Foundation (NSF), under award 201907 and the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

REFERENCES

[1] [n. d.]. An Introduction to the Intel® QuickPath Interconnect. https://www.intel.ca/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf. [Accessed: July 2023].
[2] [n. d.]. APS Upgrade. https://www.aps.anl.gov/APS-Upgrade. [Accessed: May 2021].
[3] [n. d.]. hdf5. https://www.hdfgroup.org/solutions/hdf5/.[Accessed : Septeember 2023].
[4] [n. d.]. numactl. https://github.com/numactl/numactl/tree/master.[Accessed : Septeember 2023].
[5] [n. d.]. Scaling in the Linux Networking Stack . https://www.kernel.org/doc/Documentation/networking/scaling.txt. [Accessed: July 2023].
[6] [n. d.]. Spheres Dataset. https://tomobank.readthedocs.io/en/latest/source/data/docs.data.spheres.html.[Accessed : July 2023].
[7] [n. d.]. ZeroMQ. https://zeromq.org/get-started/.[Accessed : Septeember 2023].
[8] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. 2022. Understanding Host Interconnect Congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (Austin, Texas) *(HotNets '22)*. Association for Computing Machinery, New York, NY, USA, 198–204. https://doi.org/10.1145/3563766.3564110
[9] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229
[10] Tekin Bicer. 2014. *Supporting Data-Intensive Scientific Computing on Bandwidth and Space Constrained Environments.* Ph. D. Dissertation. The Ohio State University.
[11] Tekin Bicer, Doga Gursoy, Rajkumar Kettimuthu, Ian T Foster, Bin Ren, Vincent De Andrede, and Francesco De Carlo. 2017. Real-time data analysis and autonomous steering of synchrotron light source experiments. In *IEEE 13th International Conference on e-Science (e-Science)*. IEEE, 59–68.
[12] Tekin Bicer, Jian Yin, and Gagan Agrawal. 2014. Improving I/O throughput of scientific applications using transparent parallel compression. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1–10.
[13] Tekin Bicer, Jian Yin, David Chiu, Gagan Agrawal, and Karen Schuchardt. 2013. Integrating online compression to accelerate large-scale data analytics applications. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 1205–1216.
[14] Tekin Bicer, Xiaodong Yu, Daniel J Ching, Ryan Chard, Mathew J Cherukara, Bogdan Nicolae, Rajkumar Kettimuthu, and Ian T Foster. 2021. High-performance ptychographic reconstruction with federated facilities. In *Smoky Mountains Computational Sciences and Engineering Conference*. Springer, 173–189.
[15] Paul Caheny, Lluc Alvarez, Said Derradji, Mateo Valero, Miquel Moretó, and Marc Casas. 2018. Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2018), 1174–1187. https://doi.org/10.1109/TPDS.2017.2787123

[16] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[17] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (dec 2014), 401–412. https://doi.org/10.14778/2735496.2735503

[18] Venkatarami Reddy Chintapalli, Sai Balaram Korrapati, Bheemarjuna Reddy Tamma, and Antony Franklin A. 2022. NUMASFP: NUMA-Aware Dynamic Service Function Chain Placement in Multi-Core Servers. In *2022 COMSNETS*. 181–189. https://doi.org/10.1109/COMSNETS53615.2022.9668603

[19] Yann Collet. 2011. LZ4 - Extremely Fast Compression algorithm. https://github.com/lz4/lz4. [Online; accessed 6-22-2023].

[20] Juan A. Colmenares, Reza Dorrigiv, and Daniel G. Waddington. 2017. Ingestion, Indexing and Retrieval of High-Velocity Multidimensional Sensor Data on a Single Node. arXiv:1707.00825 [cs.DB]

[21] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of Query Plans on Multicores. *Proc. VLDB Endow.* 8, 3 (nov 2014), 233–244. https://doi.org/10.14778/2735508.2735513

[22] Nathan Hanford, Vishal Ahuja, Matthew Farrens, Dipak Ghosal, Mehmet Balman, Eric Pouyoul, and Brian Tierney. 2016. Improving network performance on multicore systems: Impact of core affinities on high throughput flows. *Future Generation Computer Systems* 56 (2016), 277–283. https://doi.org/10.1016/j.future.2015.09.012

[23] Mert Hidayetoğlu, Tekin Biçer, Simon Garcia De Gonzalo, Bin Ren, Doğa Gürsoy, Rajkumar Kettimuthu, Ian T Foster, and Wen-mei W Hwu. 2019. MemXCT: Memory-centric x-ray CT reconstruction with massive parallelization. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–56.

[24] Mert Hidayetoglu, Tekin Bicer, Simon Gonzalo, Bin Ren, Vincent Andrade, Doga Gursoy, Rajkumar Kettimuthu, Ian Foster, and Wen-mei Hwu. 2020. Petascale XCT: 3D Image Reconstruction with Hierarchical Communications on Multi-GPU Nodes. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 510–522.

[25] Yang Hu and Tao Li. 2016. Towards efficient server architecture for virtualized network function deployment: Implications and implementations. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783711

[26] Intelligence Advanced Research Projects Activity. [n. d.]. Rapid Analysis of Various Emerging Nanoelectronics. https://www.iarpa.gov/index.php/research-programs/raven. [Accessed: May 2021].

[27] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access Map Pattern Matching for Data Cache Prefetch. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 499–500. https://doi.org/10.1145/1542275.1542349

[28] Taeuk Kim, Awais Khan, Youngjae Kim, Preethika Kasu, and Scott Atchley. 2018. NUMA-aware thread scheduling for big data transfers over terabits network infrastructure. *Sci. Program.* 2018 (2018), 1–8.

[29] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 555–569. https://doi.org/10.1145/2882903.2882906

[30] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview: NUMA Becomes More Common Because Memory Controllers Get Close to Execution Units on Microprocessors. *Queue* 11, 7 (jul 2013), 40–51. https://doi.org/10.1145/2508834.2513149

[31] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[32] Zhengchun Liu, Tekin Bicer, Rajkumar Kettimuthu, and Ian Foster. 2019. Deep learning accelerated light source experiments. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 20–28.

[33] Ke Meng and Guangming Tan. 2017. RING: NUMA-Aware Message-Batching Runtime for Data-Intensive Applications. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. 368–375. https://doi.org/10.1109/ICPADS.2017.00056

[34] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-Aware Data Placement and Task Scheduling for Analytical Workloads in Main-Memory Column-Stores. *Proc. VLDB Endow.* 10, 2 (oct 2016), 37–48. https://doi.org/10.14778/3015274.3015275

[35] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 22–32. https://doi.org/10.1109/PACT.2011.9

[36] Somya Singh, Tyler Stannard, Sudhanshu Singh, Arun Singaravelu, Xianghui Xiao, and Nikhilesh Chawla. 2017. Varied volume fractions of borosilicate glass spheres with diameter gaussian distributed from 38-45 micronsen cased in a polypropylene matrix. https://doi.org/10.17038/XSD/1373576

[37] Yongyu Wang. 2017. NUMA-aware design and mapping for pipeline network functions. In *2017 4th International Conference on Systems and Informatics (ICSAI)*. 1049–1054. https://doi.org/10.1109/ICSAI.2017.8248440

[38] Zeyi Wen, Xingyang Liu, Hongjian Cao, and Bingsheng He. 2018. RTSI: An Index Structure for Multi-Modal Real-Time Search on Live Audio Streaming Services. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1495–1506. https://doi.org/10.1109/ICDE.2018.00168

[39] Heng Yu, Zhilong Zheng, Junxian Shen, Congcong Miao, Chen Sun, Hongxin Hu, Jun Bi, Jianping Wu, and Jilong Wang. 2021. Octans: Optimal Placement of Service Function Chains in Many-Core Systems. *IEEE Transactions on Parallel and Distributed Systems* 32, 9, 2202–2215. https://doi.org/10.1109/TPDS.2021.3063613

[40] Se-young Yu, Jim Chen, Joe Mambretti, and Fei Yeh. 2018. Analysis of CPU Pinning and Storage Configuration in 100 Gbps Network Data Transfer. In *2018 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*. 64–74. https://doi.org/10.1109/INDIS.2018.00010

[41] Xiaodong Yu, Tekin Bicer, Rajkumar Kettimuthu, and Ian Foster. 2021. Topology-aware optimizations for multi-gpu ptychographic image reconstruction. In *Proceedings of the ACM International Conference on Supercomputing*. 354–366.

[42] Xiaodong Yu, Viktor Nikitin, Daniel J Ching, Selin Aslan, Doğa Gürsoy, and Tekin Biçer. 2022. Scalable and accurate multi-GPU-based image reconstruction of large-scale ptychography data. *Scientific Reports* 12, 1 (2022), 5334.

[43] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (jan 2019), 516–530. https://doi.org/10.14778/3303753.3303758

[44] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 705–722. https://doi.org/10.1145/3299869.3300067

[45] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-Query Optimization for Complex Event Processing in SAP ESP. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1213–1224. https://doi.org/10.1109/ICDE.2017.166