Scheduling Data Streams for Low Latency and High Throughput on a Cray XC40 Using Libfabric

Farouk Salem, Thorsten Schütt, Florian Schintke, Alexander Reinefeld
Zuse Institute Berlin
{salem, schuett, schintke, ar}@zib.de

***Abstract*—Achieving efficient many-to-many communication on a given network topology is a challenging task when many data streams from different sources have to be scattered concurrently to many destinations with low variance in arrival times. In such scenarios, it is critical to saturate but not to congest the bisectional bandwidth of the network topology in order to achieve a good aggregate throughput. When there are many concurrent point-to-point connections, the communication pattern needs to be dynamically scheduled in a fine-grained manner to avoid network congestion (links, switches), overload in the node's incoming links, and receive buffer overflow. Motivated by the use case of the Compressed Baryonic Matter experiment (CBM), we study the performance and variance of such communication patterns on a Cray XC40 with different routing schemes and scheduling approaches. We present a distributed Data Flow Scheduler (DFS) that reduces the variance of arrival times from all sources at least 30 times and increases the achieved aggregate bandwidth by up to 50 %.**

***Index Terms*—Stream Processing; Data Flow Scheduling; RDMA; libfabric**

I. Introduction

The stream processing paradigm aims at the continuous processing of incoming data at the rate of all incoming data streams sustainably. The paradigm is successfully used in time-critical applications such as sensor data processing [1], simulation and prototyping [2], and real-time query processing [3], [4]. It is especially challenging to solve when hard real-time constraints have to be met as in latency-sensitive [5], [6] and life-critical [7] applications. Some applications require to collect and aggregate stream data based on feature-, sensor-, or time-dependent constraints before they can be processed in (mini-)batches, as in case of the Compressed Baryonic Matter (CBM) experiment [8].

The mentioned CBM experiment is currently set up to explore the QCD phase diagram [9] in the region of high baryon densities. High-energy nucleus-nucleus collisions are observed by many sensors surrounding the experiment generating hundreds of data streams with an aggregate data rate of a few terabytes per second. The packets of measured data signals arrive at multiple input nodes and are analyzed in aggregated time-slices at compute nodes. To build the time-slices, each sensor contributes via its input node its measurements corresponding to that time-slice for the analysis. Subsequent time-slices are assigned to different compute nodes in a round-robin fashion for analysis to sustain the incoming data rate and to spread the load equally. The data distribution and time-slice building are done in a parallel cluster application called FLESnet, which is part of the First-Level Event Selector (FLES) [10] of CBM.

Our goal is to achieve a high aggregate bandwidth between input and compute nodes while maintaining short duration and variance of time-slice building (receiving all contributions) in a scalable system. A high aggregate bandwidth can only be achieved by utilizing all communication links at all times. Short duration and variance of time-slice building can only be achieved with uniform usage of the network. The building of subsequent time-slices is parallel in space and sequential but overlapping in time. Ensuring a uniform network utilization is necessary for achieving a sustained high bandwidth, because otherwise, some receive buffers may overflow while other contributions needed for completing other time-slice are still missing. Filled buffer space will eventually reduce the overall aggregate bandwidth, as some senders will not be able to fill their outgoing links appropriately. Indirectly, a short duration and variance of time-slice completion leads to a more regular usage of receive buffers per input link at the compute nodes, which improves the scalability of the system. Suppose we have fixed hardware components and cannot afford a linear growth of main memory for receive buffers at the compute nodes, the overall buffer space we can provide per input link at the compute nodes will linearly decrease with an increasing number of input nodes. Thus, being able to free buffer space early by low variance of time-slice completion at compute nodes is a desirable system property for scalability. In general, the buffer size determines the amount of variance a system can tolerate.

We make the following main contributions throughout this paper to achieve these goals:

- We discuss the challenges of scaling FLESnet in detail and present potential solution for its communication pattern, scalability constraints, and synchronization demands (Sect. II).
- We derive a design for a Data-Flow Scheduler (DFS) that works distributed across input and compute nodes, and describe how it addresses the scalability challenges without introducing a central component or single point of failure (Sect. III).
- To study the effectiveness of DFS, we implemented micro-benchmarks using Libfabric and MPI resembling FLESnet's communication pattern between two disjoint groups of input and compute nodes and evaluate the performance on our test machines (Sect. V-A).

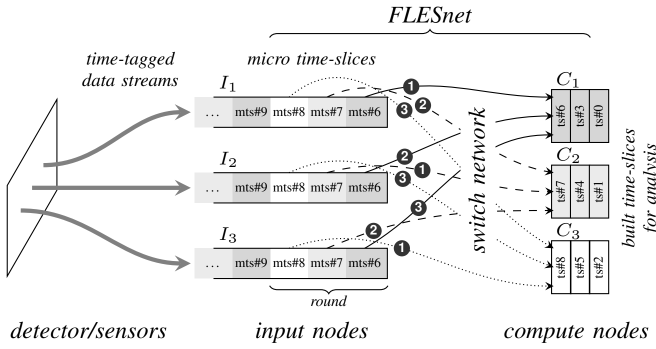

Fig. 1. A data stream flow for timeslice building.

- With DFS we demonstrate a reduction in the variance of arrival times by up to a factor of 30, an overall increase of the throughput of up to 50 % compared to FLESnet using a best-effort approach (Sect. V-B1). Furthermore, we show an improved synchronization (Sect. V-B2), a more regular usage of the receive buffers in compute nodes (Sect. V-B4), and that the overall throughput achieves at least 77 % of our micro-benchmark.

II. FLESnet Communication Pattern

The CBM experiment [8] targets to detect and discover phenomena at different times during beam-time with hundreds of sensors surrounding the experiment. Each sensor transmits the measured, timestamped data chunks to an input node, which buffers and then distributes the contributions to compute nodes (Fig. 1). To build a time-slice on a compute node for a meaningful data analysis, CBM requires each data sensor/input to contribute. The system to distribute individual data streams from input nodes to compute nodes and to build complete time-slices is called FLESnet.

FLESnet's communication pattern is mostly uni-directional and needs some buffering at input and compute nodes. Input nodes receive data streams from sensors, buffer them, chop the data into micro-time-slices (*mtss*), and scatter the *mtss* to compute nodes in a round-robin manner. Each input node distributes *mtss* to all active compute nodes in each *round*, i.e., a round consists of as many *mtss* as there are active compute nodes in the system. On the other hand, each compute node receives *mtss* as contributions for a particular time-slice from all input nodes and buffers them until the time-slice is completed when the last missing contribution arrives and the time-slice can be passed over to the data analysis.

In the previous implementation without DFS, the communication follows a best-effort approach [11] and is not well coordinated between the nodes. Input nodes distribute their round, independent of the progress of the other input nodes. Thus, compute nodes are prepared to collect several *mtss* for different time-slices from the *same* input node by allocating a part of their limited local memory as a receive buffer per input node. To prevent buffer overflow at the compute nodes, a ticket-based flow control mechanism [12] is established per pair of input and compute nodes. Input nodes receive a set of tickets from each compute node and they use each ticket to send a *mts*. Once they run out of tickets, they wait until they get new tickets when time-slices were completed (got *mtss* from all input nodes) and buffer space becomes available again. While an input node waits for new tickets, the sensors keep measuring new data, which is stored in a local buffer at the input node to avoid data loss—until also this buffer is exhausted. The lack of coordination between input nodes may lead to an unfair use of the network (everyone tries to keep sending according to best-effort), input nodes may fall far behind others only limited by buffer exhaustion, and delays finishing a time-slice occur much larger than necessary.

A. Offset-Based Round-Robin Data Distribution

Naively, all input nodes would start sending their first *mts* in the data stream to the responsible compute node—all to a single node. As this node has a limited incoming bandwidth, the compute node and the overall bandwidth would suffer from endpoint congestion [13]. All except one link to the compute nodes would remain unused during that transfer. Afterwards, the same pattern would repeat with the next set of *mtss* and the next compute node. Such an approach would effectively serialize the time slice building and would not use the available links efficiently. With the default best-effort approach of FLESnet, asynchronous send requests are issued by input nodes as long as tickets are available and *mtss* are ready to be distributed. So, for a single round the communication network is loaded with many transfer tasks ($\#inputnodes * \#computenodes$), that can be performed in parallel by the network and switches. Typically, each link of the switched network is shared for many point-to-point transfers at each time and a moderate form of endpoint congestion may still occur. With larger numbers of input and compute nodes and rounds overlapping each other, this approach may hit some scalability bottlenecks of the communication network when the number of concurrent transfers grows by factors. In addition, a single point of congestion in the network could spread through the entire network causing a tree saturation [14].

In order to evenly utilize all communication links, to reduce the number of concurrent transfers to the same receiving nodes, and to reduce the occurrence of network congestion, the input nodes should employ an *offset-based round-robin* schema, which is a combination of the round-robin schema and the index algorithm [15]. Each input node should transmit data to a different compute node at a time, as depicted by the subsequent communication steps ❶, ❷, and ❸ in Fig. 1. With this scheme FLESnet distributes the load more regularly over all compute nodes and communication links by using only a low number of concurrent connections. Instead just relying on the ticket-based flow control, FLESnet should also use a back-pressure mechanism [16] to adapt the injection rate of input nodes with the current status resp. load of the network. This would increase the achievable bandwidth when congestion can be avoided or reduced and minimize the time to complete time-slices as missing *mts* get a larger share of the

network. The effects are expected to be more pronounced with increasing number of nodes. The frequency of slow nodes and other network events will grow.

B. Scalability Constraints and Coordination

With increasing system size the overall bandwidth increases as well. However, often the local buffer space per compute node does not grow linearly with the number of input nodes. This has several effects and consequences for the system. With more input nodes, compute nodes receive *mtss* from more input nodes, but the fraction of the local buffer space per input node decreases, and consequently fewer tickets are provided to send *mtss*. In a system using eager best-effort sending and ticket-based flow control only, two trends can be expected that at first seem to be oppositional.

On the one hand, having fewer tickets per input node increases the dependency and implicit synchronisation between different input nodes as some nodes might run out of tickets and get blocked when some others cannot keep up. Such blockage happens earlier with smaller compared to larger buffers at the compute nodes, so stragglers can catch up and the network usage becomes more uniformly. As the difference between the input node with the farthest progress and the fewest progress decreases, the time to complete full time-slices should also decrease. So, scaling the system can help to implicitly synchronize it.

On the other hand, when input nodes have to be blocked, the overall throughput will not be optimal, as some links are not used until new tickets become available. The larger the system, the more input nodes will become blocked waiting for the latest contribution of a time-slice to arrive at a compute node, so they can get their next tickets. Additionally, we risk losing measurement data, when the input buffer runs full, as running out of tickets is the common case rather than the exception.

For good scalability and sustained high throughput, more coordination is beneficial. When some input nodes fall behind, the other input nodes should not try to saturate all available bandwidth to give the stragglers a chance to catch up. That way, compute nodes can collect all contributions to finish time-slices earlier, buffer fill levels are expected to remain more regular without buffers running full, and input nodes should not regularly run out of tickets, so that they can keep sending to achieve a reasonable sustained aggregate throughput. With more coordination, smaller buffers at the compute nodes may be sufficient, which would help to scale to more input nodes as smaller buffers mitigate the buffer space limitation outlined above.

Such coordination among the input nodes should happen without adding much communication/processing overhead, as that might decrease the achievable bandwidth. It should consider the network status, latency, and clock drifts of different nodes. The aim would be to steer the distribution of *mtss* so that compute nodes receive all contributions for a time-slice in a timely manner, can pass it to the analysis, and can reuse the buffer space.

C. Synchronization Aspects

To support the approaches outlined in the previous paragraphs, input nodes need to behave more synchronized in time than with the default best-effort approach. For the offset-based round-robin data distribution, input nodes should start their data distribution roughly at the same time to effectively separate the link usage. It might turn out that coordination at such fine-granular level is not possible or too costly. Then, coordination of data transfers on a round or group-of-rounds level might be sufficient when an asynchronous best-effort approach is used in between. At least, as we have discussed, it helps to assign a large share of the available network capacity to stragglers, so that they can catch up. Time-slices can be finished early and buffers are filled almost uniformly. Either, one can rely on the external clock synchronization mechanism of the system, or can integrate a clock synchronization that indirectly detects deviations and clock drift rates and provides enough feedback to each input node so that they behave synchronized.

III. Data-Flow Scheduler

Based on the considerations outlined in Sect. II, we introduce a new deterministic scheduling mechanism, called Data-Flow Scheduler (DFS), to increase the aggregated bandwidth, to stabilize the network latency, to reduce the occurrence of network congestion, and to reduce the local memory space usage for stream processing. The main contributions of the DFS are:

- It synchronizes input nodes to reduce the variance of arrival times of *mtss* for a time-slice at compute nodes, so that time-slices can be completed with shorter duration.
- It schedules the injection rate at the input nodes to avoid network and endpoint congestion, so that the overall network can be used more uniformly, which supports lower variance of arrival times.
- It dynamically adapts the injection rate trying to improve the network utilization.
- It collects and distributes coordination information decentralized among all input and compute nodes, so it is not a central component limiting scalability.

The DFS divides the transmission time into time-intervals ($intvl$). Each interval consists of a configurable number of rounds where in each round each input node transmits one *mts* to each compute node. Thereby, each compute node receives a complete time-slice per round. DFS coordinates input and compute nodes on the level of time-intervals to keep the communication overhead between different processing nodes low compared to a management per round. DFS assigns a unique identifier $intvl_{id}$ to each interval in an ascending order over time. Each interval has its meta-data, which consists of: the first time-slice to build, the number of rounds, the absolute start time, and the duration.

The DFS operates distributedly across the input and compute nodes. We call the part running on input nodes *Input Engine (IE)* and the part running on compute nodes *Distributed*

Deterministic Engine (DDE). The *IE* steers the transmission of time-intervals in each input node and collects the local contribution of the meta-data of each time-interval. The *DDE* collects the meta-data of each transmitted interval from all *IEs* and calculates the proposed meta-data for upcoming intervals. The *IE* follows the guidelines of the *DDE* and it attempts to transmit the intervals based on the proposed meta-data. Fig. 2 depicts the components of each engine.

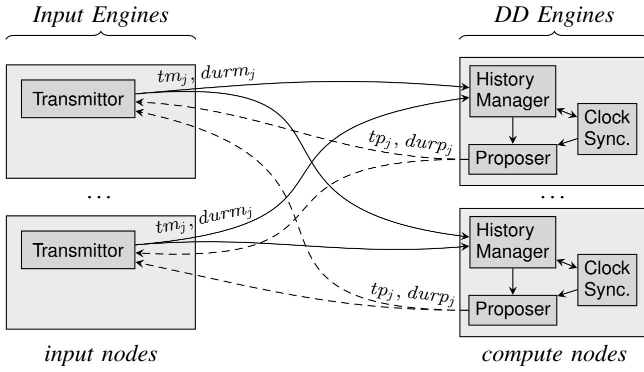

Fig. 2. DFS Engines.

A. System Assumptions

We assume a homogeneous system where all input and compute nodes are connected with the same capabilities to the network, i.e., identical network cards and network links. Otherwise, the slowest node would limit the scalability of the system or we would have to load-balance the time-slice building over the compute nodes—an aspect we plan to address in the future. The aggregated network capacity of the input nodes should be similar or smaller than the capacity of the compute nodes. The bisectional bandwidth should be close to or higher than the aggregate network capacity of the input nodes. Varying adversary traffic of colocated applications might limit the scalability. Instead of adapting to small changes, the DFS would have to constantly adapt to large changes of the available network capacity.

B. Distributed Deterministic Engine

The *DDE* consists of three modules: *History Manager*, *Proposer*, and *Clock Synchronizer*. The history manager module collects meta-data of completed intervals from input engines and calculates statistics. Proposer modules calculate the interval meta-data of the upcoming intervals based on the statistics in the history manager module with the aim to synchronize the *IEs* and to utilize the network continuously taking changes in the achievable aggregate network bandwidth into account. The proposer then broadcasts the proposed meta-data to the *IEs*. The Clock Synchronizer module compensates the different clock drift rates of the different machines to improve the synchronicity of input nodes, in case the system clocks are not well synchronized.

1) History Manager: The history manager module measures and receives the actual meta-data of intervals, which consists of the 'duration measured' $durm_{i,j}$ and the measured start time $tm_{i,j}$, for each input node i and time interval j. As different machines could have different clock speeds, it sends each meta-data to the clock synchronizer module in order to adjust it to the local machine clock. After that, it stores the actual interval meta-data of each *IE* and triggers the completion of each interval once it receives the meta-data from all *IEs*. Different *IEs* could start an interval at different times or take longer duration than what is proposed. Therefore, the history manager module calculates a unified meta-data for each interval that represents all *IEs*.

The unified meta-data of an interval j consists of the average start time avg_tm_j and the median duration med_durm_j from all *IEs*. The history manager module calculates and uses the average start time $avgtm_j$ to consider straggler nodes in its upcoming interval proposals to re-synchronize all input nodes. When an *IE* starts an interval later than what is proposed, it could be due to several reasons: (1) this input node is slow, and/or (2) one of the compute nodes is slow, and/or (3) a network link is congested. Calculating the average start time considers extreme values and delays upcoming intervals to reduce pressure on the network and synchronize the input node in time.

2) Proposer Module: The proposer module calculates the proposed start time tp_j and duration $durp_j$ of an upcoming interval j based on the meta-data of the completed intervals. As the last completed interval w could represent an extreme state of the network that does not represent the majority of intervals, the proposer module calculates the median interval duration med_durm_hist of a configurable number $hist_cnt$ of the last completed intervals. It uses this median duration to estimate the start time of the upcoming interval j. The tp_j is the completion time of the last interval w in addition to the needed duration to complete the intervals between w and j, which is calculated based on the med_durm_hist as follows:

$$tp_j = (tm_w + durm_w) + (j - w - 1) * med_durm_hist$$

The proposer module could use med_durm_hist as the proposed duration for an upcoming interval j. However, the measured duration of the last intervals could be exceptionally high due to temporary problems such as network/endpoint congestion. Therefore, the proposer module applies a staged-speed-up (*SSU*) mechanism that reduces the proposed duration of intervals in order to reach the maximum achievable bandwidth over time and to recover from temporary slow-downs. When the connections between input and compute nodes are established at the beginning, this mechanism supports to gain speed resembling the slow-start of TCP [17].

The proposer module uses the *SSU* in such a way that it does not overload or congest the network when the maximum bandwidth is achieved. The proposer module applies the SSU only when the average variance between the proposed and the measured meta-data of each interval from the last completed intervals $hist_cnt$ is smaller than a configurable

ssu_TH. A lower variance indicates synchronized nodes and a relaxed network. The *SSU* reduces the proposed duration $durp_j$ of an interval j by a configurable percentage ssu_pct, as described in Eq. 1. Once the proposer module calculates the meta-data of an interval, it transmits this meta-data to all *IEs* after adjusting it using the clock synchronizer module in order to synchronize the data transmission.

$$\mu_j = \sum_{i=w-c}^{w} |durm_i - durp_i| / hist_cnt$$

$$VAR_j = \sum_{i=w-c}^{w} (|durm_i - durp_i| - \mu_j)^2 / hist_cnt$$

$$durp_j \rightarrow \begin{cases} durp_j * \frac{(100-ssu_pct)}{100}, & VAR_j \leqslant ssu_TH \\ durp_j, & Otherwise \end{cases} \quad (1)$$

3) Clock Synchronizer: The clock synchronizer module calculates the clock offsets between the local machine and input nodes. It also tracks the clock drift of each input node over time to calculate the meta-data of the upcoming intervals correctly. When an *IE* finishes transmitting an interval j, it sends immediately the actual meta-data and the median network latency lat_i of the connection, where i is the index to indicate the ith *DDE*. The *DDE* records the local time lt when the message is received and calculates the end time of the interval $tmend_j$. Then, it calculates the clock offset for interval j as $toffset_{i,j}$ (see Eq. 2), where i is an index indicating the ith *IE*.

$$tmend_j = tm_j + durm_j$$

$$toffset_{i,j} = lt - tmend_j - lat_i \quad (2)$$

The clock synchronizer module stores the history of the clock offsets of each input node and calculates the clock drift accordingly:

$$drift_{i,x} = (toffset_{i,x} - toffset_{i,x-1})/(tmend_x - tmend_{x-1})$$

$$drift_i = median(drift_{i,k}) : \forall k \in [(j - hist_cnt) \dots j], \quad (3)$$

where j is the index of the latest finished interval.

C. Input Engine

The Input Engine receives the proposed meta-data of intervals and schedules the data transmission accordingly. Initially, when a connection is established between each pair of *IE* and *DDE*, each *IE* transmits its *mts* without any guidelines from the *DDEs*. The first intervals determine the initial duration an interval takes. After the *DDEs* receive the meta-data of the first intervals, they calculate the required duration and propose the meta-data of the upcoming intervals accordingly. When the *IE* updates the *DDEs* with the actual meta-data of an interval j, it requests the proposal meta-data of a particular interval k, where $k \geq j + 2$. The *IEs* use the last proposed meta-data when they do not receive the guidelines from the *DDEs* in time. Therefore, the DFS is a non-blocking mechanism.

When the *IE* starts a new interval, it records the local time as tm_j, divides the proposed interval duration $durp_j$ fairly on the number of interval rounds $rounds(j)$, and calculates the round start time $tr_{j,y}$ and duration $durr_j$ accordingly, where j is the interval index and y is the round index, as outlined in Eq. 4. The *IE* attempts to start each round at its proposed time and divides the duration to the next round fairly on transmitting its *mtss*.

$$durr_j = durp_j / rounds(j)$$

$$tr_{j,y} = tm_j + durr_j * y \quad (4)$$

The *IE* receives an acknowledgment when a compute node receives a contribution. It calculates the latency of receiving each acknowledgment and calculates the median $medlat$ accordingly at the end of each interval. After transmitting all contributions of a particular interval, the *IE* waits until receiving a specific threshold θ of acknowledgments before starting a new interval. If a node is slow and it reached the proposed end time of an interval $tp_j + durp_j$ without sending all the contributions, the *IE* starts transmitting the remaining contributions using the best-effort traffic. As a result, this node utilizes the network as fast as it can while other input nodes might idle. The idle time of other nodes would be kept short as this *IE* tries to catch up.

When the *IE* receives the last acknowledgment of a particular interval, it records the local time and calculates the actual interval duration tm_j. Then, it broadcasts this meta-data to all *DDEs*. Therefore, the DFS is a deterministic mechanism for scheduling data distribution because the DDEs get the same history of completed intervals and thus they propose the same meta-data for the upcoming intervals.

D. Fault Tolerance

The *DDEs* propose the same meta-data for each interval and thus are replicas to each other. The *IEs* use the first received meta-data of an interval proposal without waiting for other *DDEs*. Therefore, DFS needs an *IE* at each input node and at least one *DDE* on any of the compute nodes to be running. When a system uses more *DDEs* on different compute nodes, they tolerate a failure of any but one of them.

Different *DDEs* do not communicate between each other. When the DFS runs with a single *DDE* and it fails, the new restarted *DDE* cannot propose time-intervals immediately. It assumes that there might be other running replicas and it might propose inconsistent meta-data. In order to recover a failed *DDE* and to keep the mechanism deterministic, the history manager module of the new *DDE* has to collect $hist_cnt$ complete intervals before it starts proposing. As a result, all *DDEs* would keep proposing deterministic meta-data of intervals.

When a compute node fails, the *IE* would distribute fewer contributions per round because the *DDEs* propose the meta-data based on the history of the completed intervals that they have. However, the *SSU* mechanism would reduce the interval duration over time and the *DDEs* would propose the optimal interval duration after a set of intervals. When an input node fails, each *DDE* detects that it does not receive contributions for an interval. After that, it can propose the

upcoming intervals based on fewer *IEs*. Therefore, The DFS is able to recover the failures of both input and compute nodes. For the CBM use case, it does not make sense to tolerate failures of input nodes, as only complete time-slices are reasonable for later analysis.

IV. Implementation

FLESnet was initially implemented[1] using the API of InfiniBand Verbs [18] and it relies on connection-oriented communication. To support other modern interconnects like Omni-Path [19], Ethernet, and GNI [20], we ported[2] FLESnet to OpenFabrics Interface (OFI) Libfabric [21], [22]. Input and compute processes run on separate single cores and each process uses a single thread. In order to initially synchronize input and compute processes, FLESnet uses the $MPI_Barrier$ once the connections are established and before the start of transmitting the *mtss*. We assume that all processes leave the barrier at a similar time and most often they actually do so. Each process records the local time once it leaves the barrier and the input processes broadcast their triggered time to the compute processes. The *DDE*, which runs on compute processes, uses these times to calculate the initial clock offset of each machine. We implemented our DFS[3] on top of FLESnet.

FLESnet uses two types of messages to communicate between nodes:

- Remote Direct Memory Access (RDMA) writes [23]: Input nodes write the *mtss* into the memory of remote compute nodes using RDMA writes. RDMA is a one-sided communication, therefore compute nodes are not informed or interrupted when a *mts* is written into their memory.
- Message Passing (SYNC) messages: To coordinate between input and compute nodes, message passing is used. SYNC messages are only used when a node wants to inform another node about changes. Input nodes use this message to inform compute nodes about the written data, actual meta-data of intervals, and to request proposal meta-data of a particular interval. On the other hand, compute nodes use SYNC messages to send tickets, or proposed meta-data of upcoming intervals, and to inform input nodes about completed time-slices.

FLESnet is developed to receive variable sizes of *mtss*, therefore the buffer spaces are designed as ring buffers in order to utilize the usage of the available memory space. When the end of the ring-buffer is almost reached and there is a small space at the end and enough space at the beginning of the buffer, the input node divides the *mts* into two parts, to fit the available space, and writes the *mts* using two RDMA writes, instead of one. Additionally, each *mts* has a descriptor of 20 bytes that describes the component and content of the *mts*. This descriptor is written in a separate RDMA write after transmitting the *mts* content. After the *mts* is written, input nodes use a SYNC message to inform the compute node about the written *mts*. Once an *IE* transmitted a whole interval, it uses SYNC messages to broadcast the actual meta-data.

V. Evaluation

We evaluated the Data Flow Scheduler on a Cray XC40 using up to 384 nodes, each equipped with two Intel Xeon E5-2680v3 and 64 GiB of main memory. We use the GNI provider [20] of Libfabric to run FLESnet on Cray Aries network. We implemented a micro benchmark that resembles the communication pattern of FLESnet—half of the nodes is spreading messages to the other half of the nodes—but without any coordination or dependencies on the arrival of contributions of other nodes or buffer space limitations. This allows us to compare the performance of RDMA write operations between MPI and Libfabric to evaluate our decision of porting FLESnet to Libfabric (Sect. V-A). It also gives us an upper limit of the bandwidth that the FLESnet architecture reasonably could achieve. Then, we discuss the performance of DFS in various aspects. We show the effect of DFS on the aggregated bandwidth (Sect. V-B1), how DFS reduces the duration to complete time-slices (Sect. V-B2), the status of the buffer space (Sect. V-B4), and the effectiveness of the staged-speed-up mechanism (Sect. V-B3).

We used Libfabric 1.6.2 and Cray-mpich 7.5.1 versions compiled on Cray GCC 6.2.0 compiler. The micro-benchmark of Libfabric[4] and MPI[5] are adopted from [24], [25] respectively. We modified these benchmarks to divide the processing nodes into two groups: half of the nodes as senders, and the other half as receivers. Each processing node runs a single thread, either sender or receiver. The benchmark measures by repeated executions the average duration of a round (each sender writes a message into each receiver's memory buffer) and the average bandwidth of RDMA writes using different number of processes on a single run.

A. Libfabric/MPI Micro-Benchmark

The MPI and Libfabric micro-benchmarks use the MPI_Put and fi_write operation respectively to write data into the receiver's memory. In order to improve performance, the benchmark uses huge pages, which defaults to 2 MiB pages for the benchmark. The benchmark aligns the senders to start writing data roughly at the same time using an $MPI_Barrier$. Each sender finishes writing data independently on other senders after a given number of iterations.

The results show that Libfabric needs a longer duration to transmit a round of messages when the message size is larger than 128 KiB, as depicted in Fig. 3. MPI achieves 8.7 % and 50.65 % shorter duration for 128 KiB and 1 MiB messages respectively when 192 nodes are used. Therefore, Libfabric achieves a better bandwidth than MPI when the message size is at most 128 KiB, as depicted in Fig. 4. While MPI improves the performance for larger messages, Libfabric

[1]https://github.com/cbm-fles/flesnet

[2]https://github.com/tschuett/flesnet

[3]https://github.com/tschuett/flesnet/tree/fles_libfabric_DFS

[4]https://github.com/fsalem/cray-tests

[5]https://github.com/fsalem/mpi-benchmarks

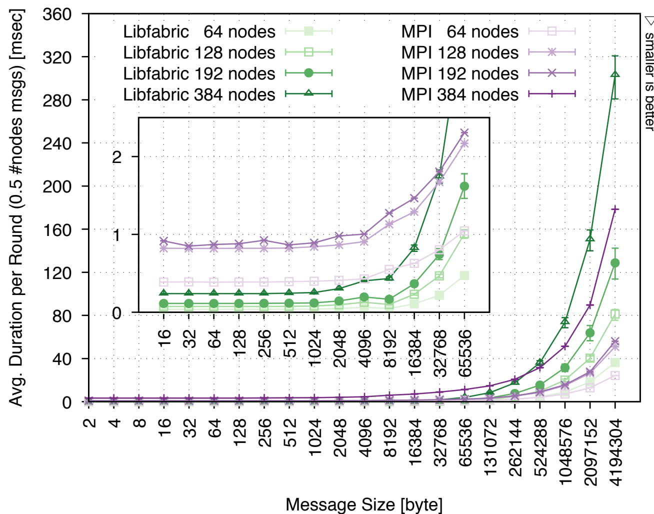

Fig. 3. Libfabric-MPI average duration per round of RDMA writes of many concurrent point-to-point communications with different number of nodes (half senders and half receivers).

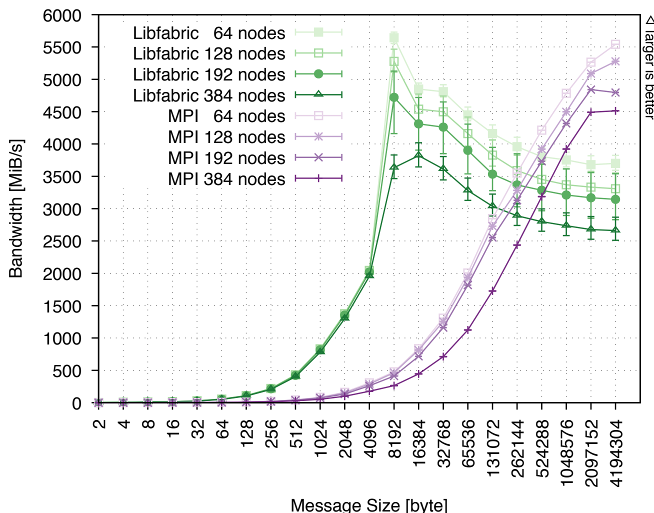

Fig. 4. Libfabric-MPI bandwidth per sender of RDMA writes of many concurrent point-to-point communications with different number of nodes (half of them sender and half of them receiver).

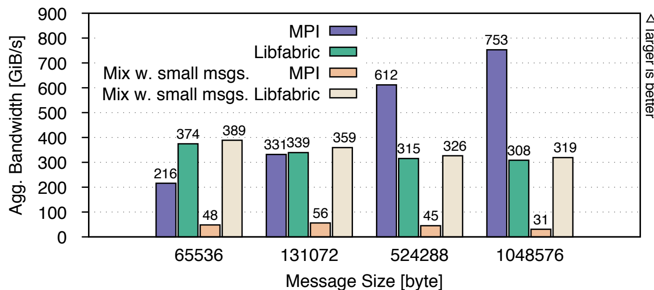

Fig. 5. A comparison between the aggregated bandwidth of Libfabric and MPI micro-benchmarks at different message sizes and a workload mixed with small messages; for each big message one 16 and two 64 byte messages are sent in the 'Mix' scenarios. No synchronization overhead is involved and each sender is sending to all receivers with point-to-point communication on 192 nodes (96 senders to 96 receivers)

TABLE I
CONFIGURABLE PARAMETERS

Parameter	Configured	Value
ts_per_intvl	10,000	timeslices; timeslices per interval
hist_cnt	10	intervals; length of history
ssu_TH	10	%; average variance threshold for speed-up
ssu_pct	5	% of the interval duration; speed-up amount

suffers from a significant performance drop with messages larger than 8 KiB. We use huge pages of 2 MiB to improve the performance of large messages. In general, it would help to use multiple threads per node [26].

This micro-benchmark only shows the achievable performance when RDMA writes are used overriding the memory space without any demand of tickets and coordination. It differs from FLESnet in several aspects:

- FLESnet transmits different types of messages with different sizes, as explained in Sect. IV, while the benchmark transmits one message type (RDMA writes) with a fixed message size.
- FLESnet has a limited buffer space at each node and tickets have to be available to write more *mtss*. To receive a new ticket for the same buffer place, input nodes have to transmit the *mts*, wait to receive the completion event of the networking layer, send a SYNC message to inform the compute node about the written *mts*, wait to complete the time-slice at the compute node (collect all contributions from other nodes), and then receive a SYNC message containing a new ticket. The benchmark, on the other hand, can send as many messages as fast as it can without these limitations.
- The nodes of FLESnet depend on each other's progress. One straggler node quickly affects other nodes, as discussed in Sect. II. In the benchmark, in contrast, senders and receivers do not depend on each other at all.

Figure 5 depicts the aggregated bandwidth of the micro-benchmark (#senders*'bandwidth per sender') for MPI and Libfabric and compares it to a scenario where a fixed number of SYNC messages and timeslice-descriptors are added per round. These results show the aggregated bandwidth of writing 1 million *mtss*, 1 million 16 byte descriptors—one for each time-slice, and 2 million 64 byte SYNC messages. We see a significant performance drop when small messages are added to the workload, especially in case of MPI. These results confirm our decision to port FLESnet to Libfabric in order to support various sizes of time-slices performance wise. We will use the Libfabric benchmark results for the workload including small messages (rightmost bars in Fig. 5) to evaluate the effectiveness of DFS.

B. DFS Performance

We configured FLESnet to assign 1 MiB of main memory to each node and each *mts* is 64 KiB. Table I shows the values of the other DFS parameters.

1) Achieved Throughput: We benchmarked FLESnet and FLESnet with DFS (short DFS) and compare the results with

Fig. 6. A comparison between FLESnet and DFS of the duration to complete each time-slice. The plots, on the right summarize the left side plots by showing the minimum, maximum, 10th and 90th percentile, and the median. Half of the nodes are input nodes and the other half are compute nodes.

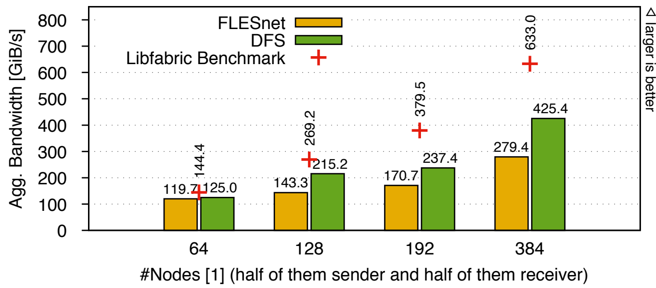

Fig. 7. Achieved aggregate bandwidth of FLESnet and DFS compared to the Libfabric benchmark with different number of nodes.

the Libfabric micro-benchmark results with a mix of message sizes (see Sect. V-A). This micro-benchmark is comparable with FLESnet regarding only the data distribution because FLESnet uses a combination of SYNC messages (87 bytes), *mts* RDMA writes (64 KiB in these test cases), and *mts* descriptor RDMA writes (20 bytes). We discuss how the system limitations of FLESnet (see Sect. II), cause a significant performance drop on larger systems.

Figure 7 depicts the achieved aggregated bandwidth of FLESnet and DFS on different system sizes. DFS shows a better performance and it almost closes the gap to the Libfabric benchmark. The DFS achieves 80 %, 62.5 %, and 67 % of the Libfabric aggregated bandwidth on 128, 192, 384 nodes respectively, while FLESnet achieves 53 %, 45 %,

44 % of the Libfabric aggregated benchmark using the same sequence of nodes. For larger systems, the synchronization overhead increases.

2) Synchronization Overhead: One of the goals of DFS was to shorten the duration to receive a complete time-slice at the compute nodes to free the buffer space as fast as possible. Figure 6 depicts the time difference between the arrival of the first and the last *mts* of each time-slice. DFS shortens the duration by at least 30 times. The figure shows that DFS (as FLESnet) needs longer to complete time-slices when the system scales up because it collects *mtss* from more nodes, and therefore it needs more time. Nevertheless, the variance of the duration to complete time-slices remains steadily short with larger systems.

3) Bandwidth Recovery: The DFS is able to recover temporary bandwidth drop, for example because of network congestion. We simulated an artificial bandwidth drop by increasing the actual duration of an interval by 25 %. Figure 8 depicts a comparison between the proposed duration and the actual duration of interval. It compares a normal running of DFS when there are no problems and when it faces bandwidth drop. The figure illustrates that the DFS recovers the dropped bandwidth over time.

4) Buffer Usage: Due to the reduced completion time for collecting a time-slice, DFS is able to free the buffer space faster than bare FLESnet. Figure 9 illustrates the buffer fill levels aggregated across compute nodes and over time. For each sample, the buffer fill level at each compute node is ordered in descending order. The figure shows that many buffers are filled up in case of FLESnet, i.e., input nodes run out of tickets. With DFS the buffer fill level is only ≈10 % for all connections. As we explained earlier, this is essential for the system's scalability because the local memory space does not scale up with the system size.

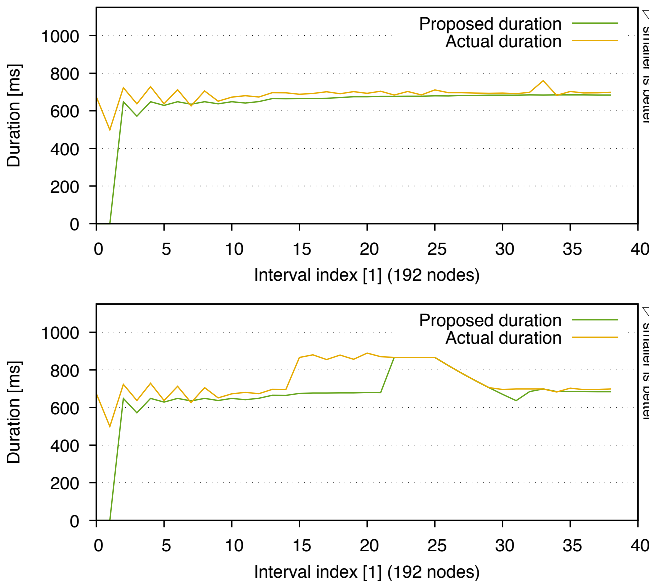

Fig. 8. A comparison between the proposed and actual duration of a sample of intervals on 192 nodes (half of them input nodes and half of them compute nodes). The top plot shows a normal meta-data of intervals when the bandwidth is stable. The bottom plot depicts the effect of the staged-speedup mechanism when a 25 % bandwidth drop is artificially applied for intervals 15 to 25 and the SSU is applied for intervals 25 to 35.

VI. Related Work

There are a variety of different measures to detect, avoid, and cope with congestion in different disciplines. A common technology is credit-based flow control [27]. It is similar to our approach and uses credits to manage the network throughput. The Aries network [28] uses adaptive routing to route around congested parts of the network.

Monitoring and Detection: On top of existing functionality in common switches, Everflow [29] designed a network telemetry system on packet-level. It distributes collected packages of several analysis servers and uses 'guided probes' to detect and analyze faults. In contrast, we decided to use the existing servers to monitor the network.

SketchVisor [30] uses counters to detect problematic flows with low overhead. Their prototype is built on top of Open vSwitch. The so called fast-path is used for more detailed analysis. It uses Misra-Gries's top-k algorithm to identify offending flows. We keep detailed information about each flow in our approach.

In contrast to previous systems, Trumpet [31] performs active monitoring of network flows. Instead of using switches to monitor traffic, they employ the end-hosts, which analyze all incoming resp. outgoing traffic. Users can deploy triggers at the nodes. It resembles our approach, as we also rely on end-hosts for analyzing the network. Our approach knows the overall communication pattern in advance and thus can plan the schedules instead of just reacting to the occurring load.

Load-Balancing: State of the art load balancers, such as Google's Maglev, maintain state per connection to provide consistency in face of backends joining or leaving. In contrast, Beamer [32] is a stateless loadbalancer. It relies on stable hashing, fault-tolerant control pane, and in-band communication between switches. When the compute nodes have heterogeneous compute capacity, we would follow a similar approach to adaptively distribute the load.

Facebook uses a distributed video processing system called SVE [33]. Similar to our approach, it is streaming based and has predictable network flows. Another practical scenario where the usage of DFS could be advantageous.

Traffic Shaping: Instead of a centralized resource for managing traffic, Carousel [34] lets the end-hosts control their data-center network. Traffic shaping includes packet pacing, rate-based congestion control, and policy-based bandwidth allocation to flows. The DFS also uses a distributed monitoring and scheduling system for scalability. Instead of reacting to the arising network load, it knows the overall communication pattern in advance and thus can plan the schedule ahead.

With an improved switch queuing algorithm, multipath routing, and a new transport protocol, NDP [35] can provide low-latency, isolation between different workloads in data-

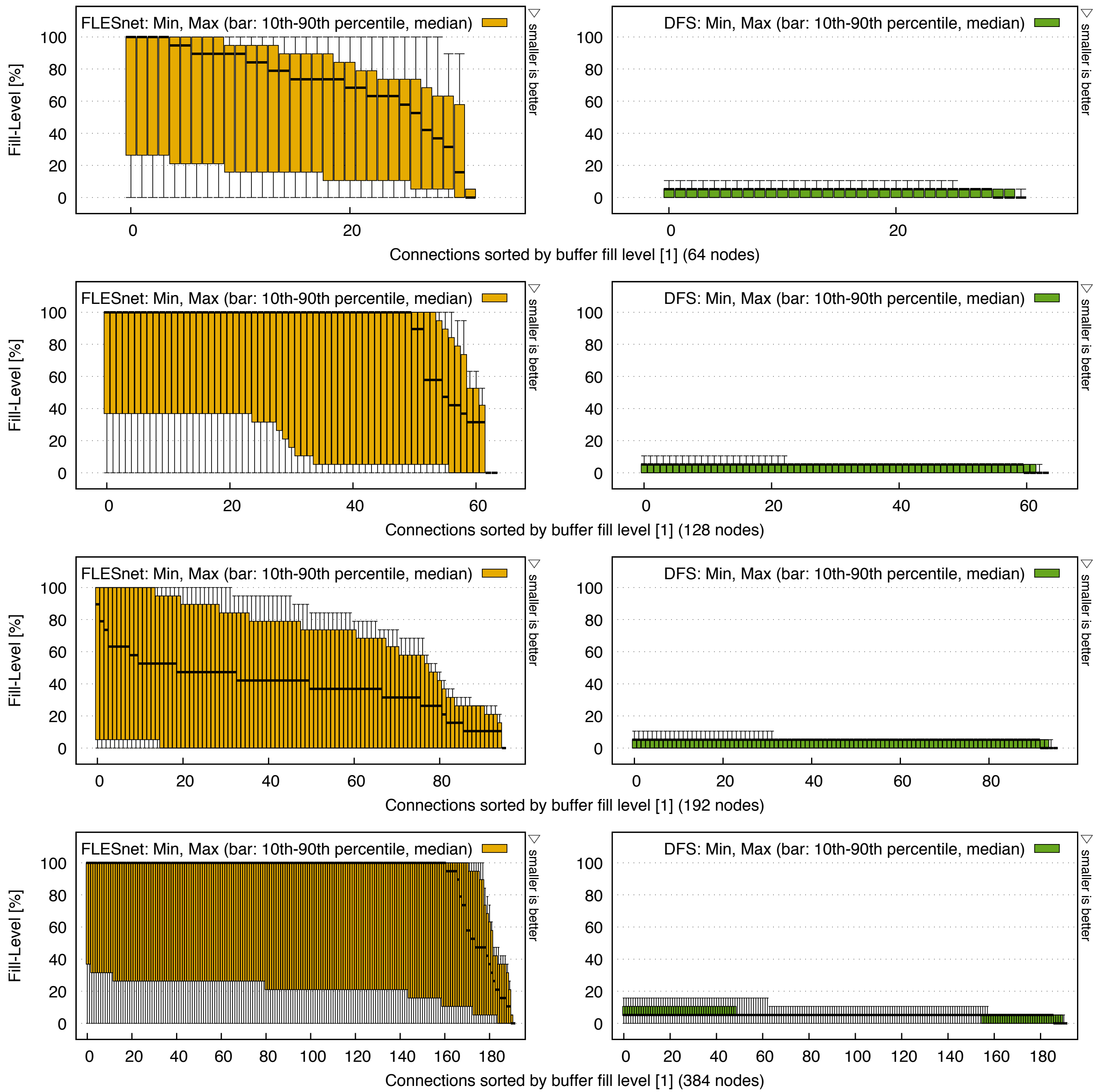

Fig. 9. The minimum, maximum, 10th and 90th percentile, and the median of the buffer fill level on ordered connections of 64, 128, 192 nodes respectively, aggregated for all compute nodes over a whole run; sampled every second. Each run took at least 18 minutes and each input node has access to a buffer space at compute nodes that can hold up to 16 *mtss*, so the overall buffer space is scaled linearly with the number of compute nodes for this evaluation. Half of the nodes are input nodes and the other half are compute nodes.

center networks, fairness, and avoid congestion. It is optimized for CLOS networks.

While Carousel follows the trend to move traffic shaping to the end-hosts, DRILL [36] follows the opposite direction. It uses switches to perform micro load balancing based on queue occupancy and randomizing the traffic. In contrast to these approaches, DFS uses an end-host based approach and works independent and on top of the underlying network routing.

Using credit-based flow control, ExpressPass [37] provides bandwidth allocation and fine-grained packet scheduling. It has similar goals to our approach: fast convergence, low buffer occupancy, and high utilization.

Similar to our approach, some systems rely on end-hosts to perform traffic shaping. Other system employ the servers to manage the network. We only rely on the end-hosts. The literature also shows some cases of using credit-based flow control, which we used to manage data flows. The typical network traffic in a data-center is highly challenging as it is unpredictable and constantly changing with micro bursts. Similar to SVE [33], our traffic pattern is more constant and predictable, which allows us to use different techniques.

VII. Conclusion

We presented a distributed data-flow scheduler (DFS) which runs on a set of senders and receivers to steer high-volume data stream distributions with a high throughput, as needed for the Compressed Baryonic Matter (CBM) experiment. DFS aims at achieving a fair network usage, so that stream chunks from the same observation time are aggregated in the compute nodes without much time delay and low buffer usage.

DFS is non-blocking, distributed, and provides deterministic data flow schedules for the senders based on the behavior observed in the recent past. Due to the use of Libfabric it works not only with Cray Aries/GNI but also with other modern interconnects. Compared to generic data-center solutions, DFS is coupled with the application and thus knows the intended communication pattern in advance, which gives DFS the advantage to be able to calculate a schedule that is given to the input nodes, so they can try to follow the schedule. This way, we outperform the integrated adaptive routing of the network, because DFS can leverage knowledge that is not available to a reactive system.

As shown in the paper, DFS improves both, the system scalability and the performance. It increases the aggregated bandwidth (80 % vs. 53 % for FLESnet of the practicably achievable bandwidth on 128 nodes) and reduces the duration to collect all time-slice contributions by a factor of up to 45. DFS synchronizes the input nodes to receive complete time-slices at compute nodes in a timely manner. As a result, the buffer space at the compute nodes is less filled (DFS needs only up to 10 %), which is essential for the system scalability. DFS distributes the load more evenly on the network resources to saturate all communication links in any time. It reduces the congestion of the network by scheduling the outgoing data packets of input nodes to different compute nodes at a time. DFS detects dynamic network changes and reschedules the traffic accordingly. It is also able to tolerate the failure of its distributed instances.

A. Acknowledgements

We thank the HLRN and Zuse Institute Berlin for computing time on the Cray XC40. The project received funding from the BMBF under grants 05P15ZAFC1 and 05P19ZAFC1 (both CBM).

References

[1] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, p. 668, ACM, 2003.

[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogenous systems," *Int. Journal in Computer Simulation*, vol. 4, no. 2, 1994.

[3] S. B. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan, "The Aurora and Medusa projects," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 3–10, 2003.

[4] S. Group *et al.*, "STREAM: The Stanford stream data manager," tech. rep., Stanford InfoLab, 2003.

[5] S. K. Barker and P. J. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems, MMSys 2010, Phoenix, Arizona, USA, February 22-23, 2010*, pp. 35–46, ACM, 2010.

[6] S. Agarwal and J. R. Lorch, "Matchmaking for online games and other latency-sensitive P2P systems," in *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*, pp. 315–326, ACM, 2009.

[7] R. Geist, M. Smotherman, K. S. Trivedi, and J. B. Dugan, "The reliability of life-critical computer systems," *Acta Inf.*, vol. 23, no. 6, pp. 621–642, 1986.

[8] V. Friese, "The CBM experiment at GSI/FAIR," *Nuclear Physics A*, vol. 774, pp. 377–386, 2006.

[9] M. Stephanov, "QCD phase diagram: an overview," *arXiv preprint hep-lat/0701002*, 2006.

[10] J. De Cuveland, V. Lindenstruth, C. Collaboration, *et al.*, "A first-level event selector for the CBM experiment at FAIR," *Journal of physics: Conference series*, vol. 331, no. 2, p. 022006, 2011.

[11] X. Xiao and L. M. Ni, "Internet QoS: A big picture," *IEEE Network*, vol. 13, pp. 8–18, Mar. 1999.

[12] L. Li and J. Zhu, "Ticket-based traffic flow control at intersections for internet of vehicles," in *IEEE International Congress on Internet of Things, ICIOT 2017, Honolulu, HI, USA, June 25-30, 2017*, pp. 66–73, IEEE Computer Society, 2017.

[13] T. E. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan, "PCP: efficient endpoint congestion control," in *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings.*, USENIX, 2006.

[14] G. F. Pfister and V. A. Norton, ""Hot spot" contention and combining in multistage interconnection networks," *IEEE Trans. Computers*, vol. 34, no. 10, pp. 943–948, 1985.

[15] J. Bruck, C. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 11, pp. 1143–1156, 1997.

[16] C. M. D. Pazos, J. C. Sanchez-Agrelo, and M. Gerla, "Using back-pressure to improve TCP performance with many flows," in *Proceedings IEEE INFOCOM '99, The Conference on Computer Communications, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, The Future Is Now, New York, NY, USA, March 21-25, 1999*, pp. 431–438, IEEE Computer Society, 1999.

[17] W. R. Stevens, "TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," *RFC*, vol. 2001, pp. 1–6, 1997.

[18] V. Velusamy, C. Rao, S. Chakravarthi, J. P. Neelamegam, W. Chen, S. Verma, and A. Skjellum, "Programming the Infiniband network architecture for high performance message passing systems," in *ISCA PDCS*, pp. 391–398, Citeseer, 2003.

[19] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel Omni-path architecture: Enabling scalable, high performance fabrics," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 1–9, IEEE, 2015.

[20] H. Pritchard, E. Harvey, S.-E. Choi, J. Swaro, and Z. Tiffany, "The GNI provider layer for OFI libfabric," in *Proceedings of Cray User Group Meeting, CUG*, 2016.

[21] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the OpenFabrics interfaces—A new network API for maximizing high performance application efficiency," in *23rd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2015, Santa Clara, CA, USA, August 26-28, 2015*, pp. 34–39, IEEE Computer Society, 2015.

[22] M. Luo, K. Seager, K. S. Murthy, C. J. Archer, S. Sur, and S. Hefty, "Early evaluation of scalable fabric interface for PGAS programming models," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6-10, 2014* (A. D. Malony and J. R. Hammond, eds.), pp. 1:1–1:13, ACM, 2014.

[23] R. Recio, B. Metzler, P. R. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," *RFC*, vol. 5040, pp. 1–66, 2007.

[24] OpenFabrics Interfaces for Cray systems, "cray-tests." https://github.com/ofi-cray/cray-tests, 2019. [Online; accessed 01-March-2019].

[25] Intel, "Intel MPI benchmark." https://github.com/intel/mpi-benchmarks, 2019. [Online; accessed 01-March-2019].

[26] D. Doerfler, B. Austin, B. Cook, J. Deslippe, K. Kandalla, and P. Mendygral, "Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, 2018.

[27] S.-A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken, "An overview of QoS capabilities in Infiniband, advanced switching interconnect, and ethernet," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 32–38, 2006.

[28] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: A scalable HPC system based on a Dragonfly network," in *Proceed-*

[36] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "DRILL: Micro load balancing for low-latency data center networks," in *Proceedings of the Conference of the ACM Special Interest Group on*

ings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, (Los Alamitos, CA, USA), pp. 103:1–103:9, IEEE Computer Society Press, 2012.

[29] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), pp. 479–491, ACM, 2015.

[30] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "SketchVisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), pp. 113–126, ACM, 2017.

[31] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *ACM SIGCOMM*, 2016.

[32] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with Beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, (Renton, WA), pp. 125–139, USENIX Association, 2018.

[33] Q. Huang, P. Ang, P. Knowles, T. Nykiel, I. Tverdokhlib, A. Yajurvedi, P. Dapolito, IV, X. Yan, M. Bykov, C. Liang, M. Talwar, A. Mathur, S. Kulkarni, M. Burke, and W. Lloyd, "Sve: Distributed video processing at facebook scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 87–103, ACM, 2017.

[34] A. Saeed, N. Dukkipati, V. Valancius, T. Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end-hosts," in *ACM SIGCOMM 2017*, 2017.

[35] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), pp. 29–42, ACM, 2017.
Data Communication, SIGCOMM '17, (New York, NY, USA), pp. 225–238, ACM, 2017.

[37] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), pp. 239–252, ACM, 2017.