A Flexible and Distributed Runtime System for High-Throughput Constrained Data Streams Generation

Paul Godard*†, Vincent Loechner†, Cédric Bastoul†, Frédéric Soulier, Guillaume Muller*

*Caldera, France
†University of Strasbourg and INRIA, France

Abstract—**Major research topics on parallel and distributed frameworks focus on reliability, performance and programmability of large scale systems for, e.g., HPC or Big Data. The solutions proposed are often directly impacted by the large scale nature of the problems. Differently, high-throughput data stream generation is an important challenge for many scientific and industrial applications which is typically well suited for small to medium scale systems, and which has to respect specific constraints about, e.g., speed, throughput or output location. In this paper we present a framework dedicated to this class of problems. We propose a performance-oriented runtime system architecture able to generate constrained data streams issued from jobs dynamically submitted by the user. Our architecture is designed to scale from a single host to a medium-sized cluster with large topology flexibility to achieve high throughput capabilities while being widely adaptive to a variety of problems. We provide experimental evidence of the ability of our framework to meet high-throughput constraints on an industrial use-case, i.e., professional digital printing, that may require tens of Gbit/s sustained output rates. We show in our measurements that our system scales and reaches data rates close to the maximum throughput of our experimental cluster.**

1. Introduction

During the last decade, the big data and HPC community achieved great results through a multitude of projects to improve data analysis and machine learning. Processing huge data streams has become particularly accessible. However, because in most cases these data streams are produced by sustained IoT devices or user activities, efficient data stream generation problems have not been studied in depths. Yet an efficient generation of data streams is the key to allow innovation in some scientific and industrial fields, such as running large physics or financial simulations outputting several data streams, or producing large and high-speed data streams which are sent to electronic devices, like industrial robots or wide digital printers.

Beyond the common goal to generate high-throughput data streams, these use cases bring on other requirements, for example on their input data, on the computation itself, or enforce constraints about the generated data streams. Not likely covered by existing frameworks, some of these possible requirements are indivisible input, controlled output speed, and output location.

The main contribution of this work is to propose a flexible, distributed, and scalable framework architecture able to efficiently perform data streams generation with respect to the constraints specified by the user, especially about speed and output location. Our framework implementation approaches the theoretical limits of our high-speed experimental cluster for the professional digital printing real-world use case.

The remainder of this paper is structured as follows. In Section 2 we present the context and the environment in which our framework can be run. Then, Section 3 describes its general design, while Section 4 gives further details about its implementation. Section 5 exposes our experiments, analyses the results and discusses the resulting capabilities of our framework. Finally, Section 6 details some related work, and we conclude in Section 7.

2. Context

This section presents the context and the runtime environment of the problems addressed by our framework.

2.1. Big Data Versus HPC

Current runtime systems addressing performance challenges belong to two main areas of expertise: big data and HPC. Both exploit parallelism to achieve scalability but through different approaches. On one hand, big data is mostly data-oriented, its specialized frameworks are designed to be easy to program, to deploy, and to scale thanks to high level programming [1], [2], [3]. On the other hand, HPC focuses on efficient computing, with specialized frameworks designed for performance fine tuning [4], [5]. This is illustrated by the *scale out vs. scale up* expression [6] which designates the ability to handle larger problem by adding storage for big data or computing power for HPC. Our framework is closer to the HPC family, to deal with intensive computation on huge but highly regular data, such as high-definition image streams.

2.2. Runtime Environment

The typical workflow of our framework is the following: a user submits constrained data stream generation jobs to the system; when receiving the jobs, it starts processing them in parallel, complying to both the available resources and the user-specified policy; finally each job generates one or multiple data streams meeting the constraints.

Our target running environment spans from one single host to a medium-sized cluster composed of a few dozens of nodes. Such small/medium scale is appropriate for integration in industrial processes which require data stream generation, hence our focus on this class of systems. To unify the resource view, we consider a single host as a cluster of one node. We suppose that all cluster nodes are interconnected using a local network, excluding complex routing rules or introducing jitter and/or fluctuating bandwidth.

We consider as unpredictable runtime events job submissions or dynamic node addition/removal. Hence our target cluster is likely to experience various events during its operating time. As a result, all the decisions should be taken at runtime, excluding strategies of offload decisions by precomputation.

2.3. Jobs

In our context, a *job* contains all the information needed to produce one or several data streams, i.e., location of the input data, constraints that the output data streams have to respect, and processing description. Jobs are submitted to our framework from an external system that we abstract as "the user" in the remainder of the paper. Our framework processes the submitted jobs in order, while allowing to execute simultaneous jobs in parallel according to the cluster available resources, to maximize their usage. If a submitted job cannot be processed due to resource limitation it will be refused and should be re-submitted later by the user, when another job is completed.

Our framework can process a variety of job workloads. However, in this paper we focus on jobs with characteristics and constraints emphasizing the relevance of our new solution, which has been designed to handle these characteristics efficiently. A typical job has the following properties: (1) a small or indivisible input making the use of a distributed file system or split input files unnecessary; (2) a high volume of data during the processing which makes task migration inefficient during the computational part; and (3) additional constraints on the output streams which are detailed in the following subsection.

2.4. Constrained Data Streams

In this paper, we consider a *data stream* as an ordered binary data flow with a start and an end. A data stream may be generated only by a single job, but a job may generate several data streams. The purpose of our framework is to handle efficiently the generation of constrained data streams. There exists various constraints, with impact from the generation process itself to the output quality. Our framework supports the following constraints, which may be specified by the user in the job description:

Output location – Specify the cluster nodes on which the user requires to receive the generated data streams;

Split – Split the generated data into multiple data streams which can be delivered to different output locations;

Additionally, to improve the quality of the generated data streams, our framework automatically ensures the following constraints:

Ordering – Deliver the data streams to the user in the same order as a sequential data generation;

Split computation – Generate the data by chunks, so the whole job can be split into tasks that continuously send data through a small buffer, not requiring to store the whole job generated data;

Constant speed – Supply the data stream to the user at a constant speed avoiding starvation, with a best effort principle;

High-throughput – Exploit the full network and computation capabilities of the cluster.

3. General Approach

We address the parallel and distributed constrained data stream generation challenge by using the common concept of *producer-consumer*. Three logical types of processes – scheduler, producer, consumer – provide to our framework a clear role separation that facilitates the management and the supervision of the cluster's resources. The *scheduler* achieves resource allocation to complete jobs. We describe it in Section 3.1. The *producers* execute tasks and output chunks of data to the *consumers*. We present them in Section 3.2. At startup, the administrator (the user) chooses how to deploy these logical processes across the cluster, matching the infrastructure and the hardware resources. Our framework offers a large topology flexibility, as discussed in Section 3.3. This flexibility is supported by a convenient communication organization depicted in Section 3.4. Section 3.5 finally discusses possible extensions to our framework.

3.1. Centralized Scheduler

To coordinate the jobs execution across the cluster we use a dedicated *scheduler* process to schedule the jobs and to assign the cluster's resources. Its primary objective is to find an efficient way to execute as much as possible jobs in parallel by taking advantage of the parallel resources offered by the cluster, while respecting the various data streams constraints.

We choose to use a *centralized* scheduler to ease the decision-making. This centralization provides multiples advantages in our context. First, all decisions are taken on a single point, avoiding latency and conflicts which can be hard to resolve. Indeed, previous researches tried to preserve the centralized scheduler model advantages in a decentralized one, but inevitably leads to complicate the decision-making process [7], [8] or requires to exchange control messages over the network, which induces network load and overheads. Secondly, a centralized scheduler gives the opportunity to have one unique point of communication for the user to interact with our framework, this substantially simplifies the interaction pattern with the environment. The scheduler is the process that we designate to be in charge of receiving the job submissions and the job control requests, like job cancellation. Finally, it also simplifies the communication path inside the framework by having one unique authority to connect and to refer to. Since we target small to medium sized clusters, a centralized scheduler will not induce much congestion, as it could on very large general-purpose clusters.

Despite those multiple advantages, a centralized scheduler has a major flaw: it is more sensitive to failures. We decided that this risk is acceptable since we target respecting high-throughput constraints as the priority: if the scheduler fails, the whole framework needs to be restarted. We mitigate the risks of failure by designing a clear architecture of our framework and addressing medium clusters, where the failure rate is low compared to exascale clusters [9].

To schedule multiple jobs over multiple nodes of the cluster, our scheduler uses the common strategy to split each job into smaller *tasks*, executed in parallel on different nodes. In order to be able for any node to process a *chunk* of data we introduce three types of tasks. These tasks are executed by the producers and are named *preload*, *compute* and *expel* in accordance with their function. The first task, preload, downloads the required input data to compute a given job. The second task, compute, processes a chunk to produce output data. The last task, expel, sends the produced data to the output location node(s). Beyond these three essential tasks, more types of tasks with complementary functions can be introduced to meet the nature of the covered use case.

The execution of a job consists in assigning an initial preload task to different nodes and, when they become ready, assigning them progressively the compute and expel tasks in order to complete the job. Thus, for a node, one single preload task occurs for the whole job, followed by multiple compute and expel tasks.

Among the wide variety of scheduling strategies existing in the literature (see Section 6), none of them has shown its ability to perform optimal decision on every use case and constraints. Thus, we chose to design our scheduler to support customizable scheduling policies in order to offer maximal flexibility to implement the best scheduling strategy depending on the context. The scheduler policy that we used in our experiments is presented in Subsection 4.1.

To manage the different constraints associated with each job and their generated data streams, our scheduler monitors the nodes and the status of the jobs and their tasks. This way, our framework is able to take judicious decisions at runtime, like creating the needed task(s) on the right node(s) at the right time, to ensure the constant and controlled output speed, or to supervise network load-balancing by indicating which network interface to use depending on the current workload.

3.2. Producer - Consumer

The role of the producer is to receive the tasks assigned by the scheduler, to execute them in accordance with their implementation and to notify the scheduler back for each finished task.

This way, the producer, as its name suggests, is the process in charge of producing the data of the streams. The chunks of data are issued from the different tasks assigned by the scheduler when decomposing the job. Our general approach allows by design a producer to run on whichever hardware (e.g. CPU, GPU, FPGA), as long as the producer process implements the different tasks to use it. Given that a producer is a process without any hardware assumption, it can be run on a whole node (including the CPU and the GPU resources) or it can be specially dedicated to run on only one hardware resource (e.g. a single CPU core) and there can be as many producers as there are resources on the node. These weak assumptions on producers provide our framework a wide flexibility to process efficiently a large range of problems.

The execution of a task is at the discretion of the assigned producer, depending on its implementation and configuration. For example, a producer running on a whole node can choose to run a specific task on a specific CPU core, or on a GPU; to run multiples tasks sequentially or to run them in parallel.

The consumer's role is to abstract the complexity of the running framework to the user. It is the logical location on which a data stream is outputted and where the user will access it. Its main purpose is to order the data it collects from the different producers. This design provides ordered data streams to the user independently from which producers compute them, and allows our framework to optimize their conveyance to the requested output location nodes.

To allow the scheduler to take the best decisions, the producers and the consumers announce their characteristics to the scheduler when starting. These characteristics are, among other minor parameters: their physical host id, their network connectivity characteristics, and their computation capacities (for the producers). Using these characteristics, the scheduler is able to build a reliable view of the cluster both in network and computation capabilities.

The knowledge of the cluster connectivity enables to perform a judicious communication distribution. Indeed, the scheduler is able to provide to producers and consumers the source and destination points (e.g., the network interfaces) to use if multiple paths are available, which enables to perform load-balancing among the different jobs.

Our framework supports adding and removing producers and consumers during its execution: they can dynamically join and quit the cluster, which adds flexibility in the cluster composition. Following these events, the scheduler modifies its internal view of the cluster and takes appropriate decisions by reassigning uncompleted tasks or aborting jobs. In the same way, our framework is able to react appropriately against producers or consumers crashes or disconnections.

Quitting the cluster for a producer while tasks are running on it will make the scheduler reassign those tasks to other producers. It risks to abort the job if the tasks cannot be computed by another producer in time. Quitting the cluster for a consumer while a job is associated to it will systematically abort the job since it is impossible to honor the job requested output data stream location.

3.3. Flexible Topology

As our framework offers to support parallel job execution and output locations, it implies to support at runtime a high level of flexibility and configuration of logical topology.

The scheduler selects the best producers to generate the data streams and outputs them on the mandatory consumers according to the job specification. Depending on the situation, selected producers and mandatory consumers can be on the same node or on distant ones. This characteristic introduces the notion of *localhost* and *distant* connection between a producer and a consumer depending on the location of the two processes.

In order to offer high throughput, multiple producers can serve one single consumer. Indeed, assigning successive chunks of a job to multiple producers provides the ability to sum the compute capabilities by executing them in parallel. Also, multiple consumers are required to handle the capability of splitting a data stream computed by one or multiple producers to several output location points. This relationship between producers and consumers are what we name the *logical topology*. Logical topologies can be classified into four categories as shown in Figure 1: *one-to-one*, *one-to-many*, *many-to-one* and *many-to-many* depending respectively on the number of producers and the number of consumers.

Our framework is designed to provide a total flexibility between the different logical topologies. Furthermore, it can mix all of them in the same cluster since multiple jobs can be executed in parallel across a same physical cluster while respecting their outputted data stream constraints.

The interaction between the producers and the consumers is related to the common publish/subscribe design pattern [10]. But some important differences must be mentioned. First, the subscription part is not issued by the consumer but initiated by the producer which is instructed by the scheduler to send his computed data to the consumer. Secondly, contrarily to the pub/sub design pattern, we do not propose a duplication of a data stream to multiple consumers, since it is not the purpose of our framework.

3.4. Organization and Protocol

To achieve high throughput on the outputted data streams and good performance over hardware cost ratio, our framework exploits a clear synchronization across the different processes presented in this section and in its interaction with the user.

High throughput should not be achieved at the price of high latency. Indeed, increasing the latency of a system to periodically improve the throughput usually leads to decreased global performance, due to the additional overhead introduced by the lack of responsiveness of the system.

To address this challenge, we decompose our framework in two parts. As shown in Figure 2, we distinguish a low latency control and synchronization path (left-hand part), and a high throughput pipeline of data (right-hand part). The low latency message path drives the high-throughput data path. High-throughput requires the usage of buffers [11], that are as small as possible to limit the output data latency, but large enough to ensure an efficient overlapping to hide parallel processing synchronization overhead.

To maximize the throughput, we design a large pipeline from the initial storage of the input data to the outputted data stream to the user. This pipeline is achieved by limiting to the strict minimum the synchronizations between the different processes and offer a natural synchronization.

To expect good performance in a cluster, the communication protocol between the processes is critical. First of all, the user submits a job to the scheduler, which checks the job validity and ensures that the requested consumers corresponding to the output locations are present in the cluster. If all the consumers respond to be able to handle the job (there is no failure at allocating local resources), and if there is enough producer computing capacity, the scheduler sends the user a job acceptance message.

Then, when the user informs to be ready to read produced data on the consumers, the scheduler starts scheduling job chunks over the dynamically selected producers by creating and assigning the needed tasks, until all the job data is fully sent to the consumers to generate the data streams.

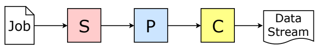
Finally, when the user has done reading all the data of the streams, it explicitly informs the scheduler that the job is completed. The scheduler will forward this information to the concerned consumers to release their resources.

This communication protocol leads to a very clear interaction between the user and the framework. All connections are specified and follow a unique path. The interaction between the user and the consumer is reduced to a simple read-only data access. This unambiguous protocol is a prerequisite to be able to isolate and handle errors or unpredictable events such as a crash or a network failure.

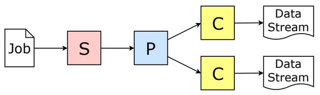
3.5. Possible Extensions

This paper focuses on presenting an effective solution to constrained high-throughput data streams generation. Its clear design and role separation allow our framework to be easily enhanced, as discussed hereafter.

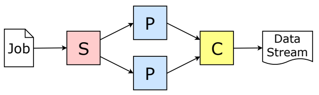
The risk of centralized scheduler failure can be mitigated by various strategies. We propose to define or elect, when the framework starts, one or multiple nodes as *backup scheduler(s)* which stay up-to-date thanks to data replication.



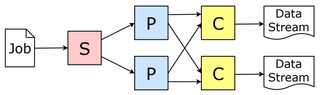
(a) one-to-one



(b) one-to-many



(c) many-to-one



(d) many-to-many

Figure 1: Logical topologies: from one-to-one to many-to-many with scheduler (S), producers (P) and consumers (C).

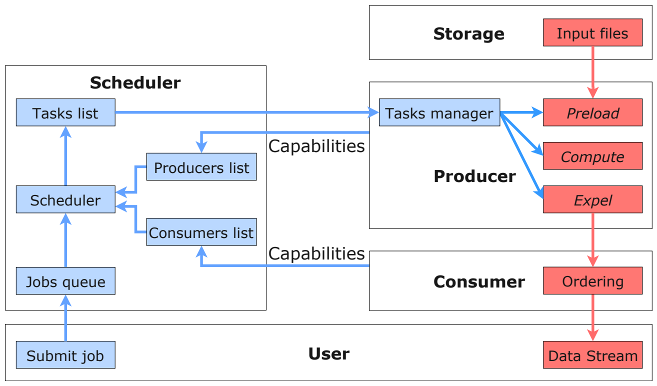


Figure 2: Overview of the low latency control path and the high throughput data path (right-hand side boxes) between the scheduler, storage, producer and consumer.

When a scheduler failure is detected, the backup scheduler replace the current one by informing all processes of the cluster and continuing its work.

To extend the use cases of our framework, we propose to support DAG (directed acyclic graph) execution by stacking and linking multiple stages of producers but keeping one unique level of output consumers. Therefore, we can support more complex scenarios including data dependencies while preserving the advantages of one consumer output stream: localization, stream delivery abstraction, controlled speed, and automatic ordering.

Finally, the computation capability of each producer could be tackled by the scheduler over the different processed tasks to be updated at runtime. Fine-tuning with real tasks execution shows its effectiveness to obtain realistic processing power estimation [12], [13]. The computation capability is exploited by the scheduler to select the best producers and create tasks which fit ideally on the selected producers. The precise knowledge of processing capabilities improves the support of heterogeneous producers [14].

4. Framework Implementation

This section provides further details about the general approach integration into our framework.

4.1. Scheduler

The scheduler is the central brain of our framework: it is responsible for coordinating the work of all producers and consumers, as well as ensuring the flexibility and a fair resources usage of the cluster to execute several jobs in parallel. While a non-optimal decision can be harmless in a small cluster with a low workload, a bad decision can result in important slowdowns in a larger cluster running at high workload.

As presented in the previous section, our scheduling policies are customizable to support specific computation problems and constraints and improve support for specific producers or consumers. In the scheduler currently implemented for our experiments, we choose a classical yet efficient load-balancing algorithm. It assigns tasks sequentially to unused producers, and it also favors locality between producers and consumers to maximize throughput and minimize latency.

The scheduler has a list of available producers and a list of output locations required by a job. By combining these two lists it can take a relevant placement decision, choosing the number and the location of the needed producers, and thus a topology among one-to-C or many-to-C (C being the number of output streams). Of course, the bandwidth is substantially improved by choosing a producer and a consumer on a same physical host.

To ensure a fair load-balancing with heterogeneous processing times, the scheduler dynamically creates tasks to decompose a job progressively. This leads to an *on-demand* policy. As soon as a producer completes a task the scheduler assigns it a new one. We favor as much as possible the presence of at least *two* compute tasks on each selected producer. This strategy allows to maintain a continuous processing by overlapping communications with computations.

The task sizes depend on the producers capabilities to reach good computing performance while keeping reactiveness of the whole system. To achieve this objective, our scheduler uses the computing capabilities announced by the producers.

As a single point of decision, our centralized scheduler is competent to receive multiple concurrent events from different producers and consumers or job submissions from the user. These events are: *join / quit* events; *job submission / job control events*; and *task feedback events*. These different events must not generate conflicting decisions. To achieve this requirement we implemented a unique event queue in which the events are placed atomically. This enforces upcoming events from any producer or consumer to be treated sequentially by the scheduler. The main scheduler process loops over this event queue.

The number of producers, consumers, and tasks in the cluster relates to the number of events managed by this atomic event queue, which may become a bottleneck. To avoid this, the tasks sizes must be large enough in order to keep the number of event per job to a moderate value. Also, if it becomes critical, some optimizations could be implemented (e.g. cache, red–black tree) or we could use a parallel event pop on non-concurrent events (e.g. events on different jobs) to reduce the event processing time. We did not meet this problem in our experiments.

4.2. Producer

The purpose of the producers is to execute as fast as possible the tasks assigned by the scheduler and acknowledge their completion.

Each type of task (preload, compute and expel) is executed by a dedicated *task engine*. Our framework is designed to run the task engines in parallel, as system threads. This leads to overlap communications by computations and thus to optimize throughput. When a task is done, the task engine sends a *task completed* event to the scheduler.

Additionally, multiple preload and expel tasks are executed simultaneously as new threads. This is particularly efficient to process in parallel multiple expel tasks associated to different network interfaces. Conversely, only one compute task is executed at a given time, since we considered that a single producer can make full usage of its available computing power.

To simplify the producer mechanisms, we designed the tasks to be run in a stateless mode. Generating and executing standalone tasks helps to maintain runtime flexibility, by suppressing the need to keep track of the previous executions.

While being very common, the preload task execution is very dependant on the cluster architecture, like the presence or not of a dedicated NAS to read an input file for example. Thus, the preload task content should be adapted by the developer to his storage solution and his input requirements. In order to provide a universal minimal solution, we implemented for our experiments a very simple NAS running in the scheduler itself as a service. It allows all producers to retrieve simultaneously an input file from the scheduler file system.

The content of a compute task must be implemented by the developer depending on the problem specification. The developer gets as input the parameters of the task created by the scheduler. Once the task computation is done, the framework associates an *id* to the data produced, which is sent to the scheduler in the task completed event.

The execution of the expel task is fully handled by the framework. When a producer starts an expel task, it is able to identify the requested data thanks to the *id* issued by the compute task. The first step to execute an expel task is to connect to the target consumer(s). In one-to-many and many-to-many topologies, the producer runs one thread per target consumer. At a second step, the producer sends to each consumer a small control message containing the range of data that will be sent to it. When a consumer is able to receive the data properly, it informs the producer to send it. Finally, the producer informs the scheduler that the expel task is completed when all data is sent.

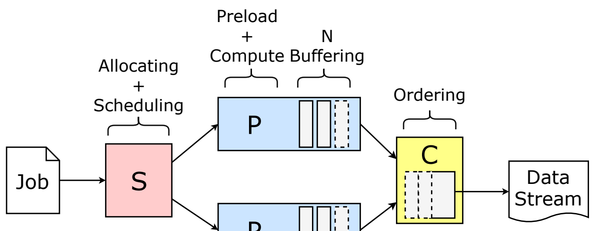


Figure 3: Data path view in the many-to-one topology with scheduler (S), producers (P), consumer (C).

4.3. Consumer

The consumer is in charge of collecting and synchronizing the different chunks of data generated on the producers into an ordered data stream delivered to the user.

The consumer will retrieve data from the producers as soon as it is available but will deliver it to the user in order. Thus, a consumer is able to block data delivery until all previous missing data has been received.

To collect efficiently the chunks of data, the consumer uses a buffer receiving the chunks in parallel as illustrated in Figure 3. If the buffer is large enough, then it is possible for the consumer to receive simultaneously different data ranges from different producers: the buffering allows to start receiving out-of-order chunks. This improves the synchronization of heterogeneous producers by relaxing the pressure in the communication between producers and consumers, in particular when they are associated to multiple jobs.

It is important to configure the buffer size wisely: while a buffer with a minimal size will help to keep low latency in the data path, a too small buffer is likely to create starvation because of data contention. At the opposite, a large buffer improves the system to be starvation-free and leads to optimize bandwidth usage, but rises the latency in the data path and increases the producer memory and cache usage (memories are shared with other producers and consumers that may run on the same node).

The ideal buffer size of each data stream on each consumer is set by the scheduler which can take a decision with a clear view of the global memory consumption, the job speed constraints, and the possible earlier starvation events in previous jobs. To maximize parallelism and communication overlapping, the circular buffer size should be at least the maximum number of producers multiplied by each producer chunk size.

4.4. Circular Buffer

We choose to use a *circular buffer* to implement the consumer data synchronization buffer. There is one circular buffer per data stream to deliver.

As the buffer is the key piece of the consumer performance, it is important to design it wisely. The circular buffer allows to manage an infinite data stream in a fixed memory space avoiding several `malloc`'s and `free`'s.

Conforming to our general approach (Section 3), the user accesses the generated data stream through the consumer, on the same node. This architecture choice allows to place the consumer circular buffer in a shared memory; and the user to directly access data in this buffer. In this way, we avoid making a supplementary internal memory copy between the two processes. The user requests access and releases blocks of data through a simple API with minimal synchronization.

To know which shared memory corresponds to which job, the circular buffers are identified by a unique *id* on the node. When the scheduler informs a consumer to become an output location for a job, the consumer allocates the circular buffer and pulls up the corresponding key to the scheduler. The scheduler then forwards this key through the job acceptance message to the user, who will use it to interact with the consumer.

4.5. Network

An efficient distributed system depends on efficient network usage. We developed a specific communication library to ensure flexible interconnection between the three types of processes in every topology, fulfilling the low latency and high throughput requirements.

Our communication library is accountable to open and manage connections over the well-known Berkeley sockets: the localhost connections with UNIX sockets, and the distant connections with INET sockets. The distant connections are handled by using the common TCP/IP network stack. It gives to our framework the flexibility to run over wide physical and logical network configurations. The usage of UNIX sockets maximizes throughput of localhost connections by avoiding the costly loopback kernel network stack.

This low level of interaction and configuration permits to fully exploit all network interfaces present on each node. In particular, we can explicitly select an input and output physical interface when opening a connection.

Our library distinguishes two types of communications to be able to perform different optimizations. The first one is the message connection dedicated to send control messages, requests and answers with low latency. The second one is the raw connection designed to provide an efficient way to send large amounts of data at full link speed.

To maximize TCP performance we take advantage of its design and its various options offered by the Linux kernel. First of all, to reduce the latency drastically when sending small messages, we disable *Nagle's algorithm* [15] and enable *quick ack* to disable the piggybacking [16]. Secondly, we activate the New Reno congestion control algorithm in order to speed up the packet retransmission, which works very well on a local network with a low RTT [17].

A persistent control connection is kept open between the scheduler and each connected producer and consumer. It is used by the scheduler to send job control messages to the producers and consumers, and by them to poll up runtime information and events. Similarly, we favor persistent producer to consumer connections. Persistent connections avoid packet losses when establishing new connections, that occur often over a highly used physical link. Once a connection is opened, it stays open and remains in a list of available connections.

Since producers send data while consumers receive data, a producer and a consumer can exploit simultaneously a full-duplex network interface for different jobs without conflict.

5. Experiments

This section provides details about our hardware and configuration setup, and presents our experiments in an industrial use case. We focus on three aspects: first, we confront our system capacity to the physical limits of the cluster; second, we evaluate our system in various scenarios of our real-world use case; third, we observe the throughput variations when enforcing a constant reading speed of the outputted data streams.

5.1. Digital Inkjet Printing Application

Complementary to pure performance evaluation, we experiment the professional digital inkjet printing as a real-world use case. This industrial application covers all the topologies through different scenarios.

The digital inkjet printing industry needs to operate several computations on data before sending them to the printer. These computations are denominated as *RIP*, for *Raster Image Processor*. A RIP converts an input image to rasterized data, which indicate where on the sheet and which ink (or color) a printer should drop ink drips. To produce this output matrix, the input image is scaled, linearized, limited and arranged. These computations introduce data dependencies between the different color layers (or channels) of the image.

Digital inkjet printing can perform a wide variety of print jobs that we illustrate by the three jobs presented in Table 1. These three jobs cover the range of data volumetry present today and in the coming years in the digital inkjet printing industry. This table specifies the print job and the volumetry of one color channel. The diversity of the digital printing industry offers a use case for each topology:

one-to-one – a single dedicated RIP engine driving a single printer;
one-to-many – a single RIP engine driving several small printers;
many-to-one – multiple RIP engines combining their computing capabilities to drive a single fast printer;
many-to-many – multiple RIP engines combining their computing capabilities to drive a high performance printer with multiple entry channels or a printer farm.

To integrate this use case into our framework, we map the following concepts. The jobs are *print jobs* which designate the printing of an image on a printer. The outputted data stream is the printed bitmap ordered by line. The preload task retrieves the input image on the producer. Finally, the consumer output location is the node where the printer driver is physically installed.

	Job A	**Job B**	**Job C**
Use case	*Low quality packaging*	*Standard quality poster*	*Very high quality banner*
Sheet size	10×10 in.	40×40 in.	100×100 in.
Sheet resolution	300×300 dpi	900×900 dpi	$1,200 \times 1,200$ dpi
Channel size	**72.0 Mbit**	**10.4 Gbit**	**115.2 Gbit**

TABLE 1: Jobs' specifications and their output matrix size for one color channel

We limit the chunk size to a fixed maximum: 200 Mbit of output data has been empirically determined to be the best trade-off for our experimental digital printing application on our cluster.

On our producer, we implement the compute task to use all available cores on the node, hence only one producer runs per node. There is also only one consumer per node, with a circular buffer of $maximal\_compute\_task\_size \times number\_of\_selected\_producer \times 4$ (4 is used to create a multiple buffering of factor four) bits, and the maximal buffer size is capped to 1 GB.

5.2. Hardware Setup

Our cluster is composed of five identical nodes equipped with an Intel Xeon D-1521, 2x 4 GB of DDR4 RAM running at 2133 MT/s, and 2x 10 GbE ports powered by a Supermicro X10SDV-4C-TLN2F motherboard. The five nodes are running Ubuntu 18.04.1 LTS with Linux Kernel 4.15.0-43-generic. They are interconnected by a dedicated Netgear XS512EM switch which provides twelve 10 GbE ports. All nodes are directly connected to the switch.

The two 10 GbE ports on each node allows to simulate up to eight distant nodes by dedicating one 10 GbE to each simulated node. We do not observe specific disturbance by simulating nodes with this strategy. It provides a realistic way to increase the number of nodes by avoiding modifications at the network stack level.

5.3. Theoretical Limits

We performed our measurements in four different topologies. Depending on the topology, the theoretical limits of the cluster are driven by memory performance, network capacity or a mixture of both.

The **one-to-one** topology exists in two configurations which implies different limits. The first one is the fully localhost one-to-one, which does not make use of the network. Hence, the limitation is the memory bandwidth to send data from the producer to consumer. It is the dual-channel RAM bandwidth[1] divided by 2: 136 Gbit/s. The second configuration is when producer and consumer are on distant nodes, thus the limit is the 10 GbE interconnection speed. As we use the TCP/IP stack over an Ethernet link, the effective bandwidth is a bit lower: 9,466 Gbit/s.

We consider the **one-to-many** and **many-to-one** topologies with distant relationship between some nodes on the cluster and the "one producer" in the first case, and the "one consumer" in the second case. Thus both topologies are limited by the link going from and going to the "one". As in the distant one-to-one topology, the theoretical limit is the network capability: 9,466 Gbit/s.

The theoretical limit of the **many-to-many** topology is the most complicated due to the mixture of localhost and distant communications between the producers and the consumers. We chose to evaluate a typical use case where each node runs both a producer and a consumer, which allows a part of the data to be transferred locally. Hence, the theoretical limit is the amount of data exchanges over the network minus the amount of data produced on the localhost producer, since the distant transfers overlap the local ones. Considering a homogeneous computing power on the producers and a homogeneous network, the limit is $N \times \frac{C}{C-1} \times H$, where N is a network link capacity, C the total number of consumers and H the number of nodes.

1. Memory bandwidth is usually expressed as GB/s, but for clarity we chose to unify all results using the common unit of Gbit/s.

5.4. Measurements

To evaluate the performances of our solution, we measure the output data rates on the different topologies for the three jobs presented in Table 1. Three input files are preloaded (for a total weight of 80 Mbits) by each producer for each job to compute.

In Figure 4, we disable the computation in the compute tasks to measure the maximal communication performance that our framework can achieve. These results are displayed against the theoretical limit of our cluster (right-hand bar of each group of bars). Complementary, Figure 5 shows the same experiments with the raster image computations enabled. The throughput is calculated by dividing the job output data size by the total elapsed time, from the job submission to the job resources release.

As explained in Section 5.2, we use the two 10 GbE interfaces of our cluster nodes to simulate up to two producers and two consumers on a single physical node. The measurements employing multiple simulated producers or consumers are annotated with an asterisk* in Figure 4.

To suppress the interferences of localhost and distant connections between the producers and the consumers, we performed our measurements on one single distant node in the one-to-many and many-to-one experiments. For the many-to-many topology measurements, each node runs both a producer and a consumer to keep a constant ratio of localhost and distant connections in the experiments, and we do not use the simulated hosts.

The one-to-one, one-to-many and many-to-one experiments run each job described in Table 1 with four color channels. In the many-to-many topology, to keep a constant workload between the nodes, each consumer is in charge of one channel: the jobs are running with as many channel as the number of nodes present in the experiments.

We evaluate the capability of our framework to deliver sustained data streams by making the user read the successive image rows at constant intervals. If the data streams rates are not sustained, we will observe starvation and a lower than expected throughput.

5.5. Results

Figures 4a, 4b and 4c show the capability of our framework to meet the theoretical limit for jobs B and C in respectively distant one-to-one, one-to-many, and many-to-one topologies. The more producers and consumers, the more stable are the results. For the job A, poor performance is due to the very small job size: the submission overhead cannot be compensated by the efficient data computation and transfer time. But in Figure 4b this overhead almost disappears due to the different consumer jobs overlapping, and every job including job A reaches the theoretical limit with 4 and more consumers. In Figure 4a the localhost one-to-one topology stays below the RAM theoretical limit due to concurrent memory operations occurring over the system, but we reach a satisfactory 90 Gbit/s on this intensive and competing in-host memory-copy operation. For the many-to-many topology in Figure 4d, we do not reach the theoretical limits. We are still investigating this issue, but we suspect network congestion: the network driver handles more connections per node, the risk of packet collisions is higher, and the switch gets loaded with the growing total number of connections and packets to commute.

Our use case experiments (Figure 5) show a great stability and scalability when the execution is computation-bound. Figure 5a shows that we get almost identical results for localhost and distant connections in the one-to-one topology: our framework can distribute producer and consumer without slowdown as long as the compute capability can be handled by the network link capacity. In the same way, Figure 5b illustrates that one producer can distribute its computation power to multiple consumers without performance loss. The many-to-one topology (Figure 5c) shows the capability of our framework to sum up the computation power of different producers all the more as the job size is large. For job A, the high latency over computation time ratio explains the absence of speedup. A similar behavior can be observed for the many-to-many topology in Figure 5d, which demonstrates the scalability of our framework. On this real-world use case, we reach more than 9 Gbit/s output rate, which is the computational limit of our experimental cluster. This output rate could probably be increased, up to the networking limits of the cluster met in Figure 4, by adding more nodes to the cluster or adding computing power to each node.

Our experiment presented in Figure 6 shows the throughput when reading the outputted data streams at constant speed. These results illustrate the capability of our framework to maintain a sustained data delivery rate at the cost of only 3.2% producer power loss on average. The worst case of 13.5% power loss in the localhost one-to-one topology results from a side effect of an active loop on the consumer side (to wait for a small time interval) and processing the data generation on the producer side on the same node; this phenomenon is also slightly visible in the many-to-many topology.

Not shown in these graphs, the standard error does not exceed few percents in worst cases (with an average of 0.5%). To conclude this experiment, we observe only a few percents power loss compared to the user requested throughput. In that respect, we expose the capability of our framework to deliver sustained data streams, at the price of a very small producer power over-allocation.

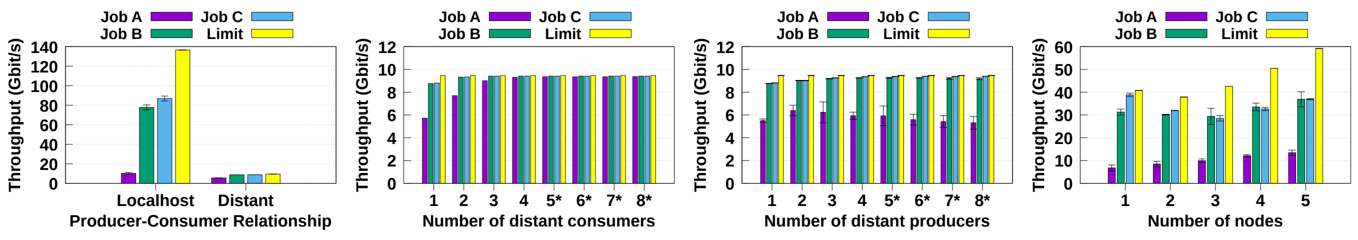
These different measurements illustrate the flexibility of our framework, by being functional on a variety of topologies, while offering a high throughput and a nice scalability and constancy. Its capability to approach the theoretical limits of the cluster permits a real-world application to fully exploit the cluster computation and communication power.

6. Related Work

Distributed systems are very common and widely used in various scopes of applications, from research to industry. Depending on the main objective of the distributed system, we can distinguish two areas of expertise: *HPC* were pure computing performance comes first, and *big data* where the capacity to process extremely large data sets comes first.

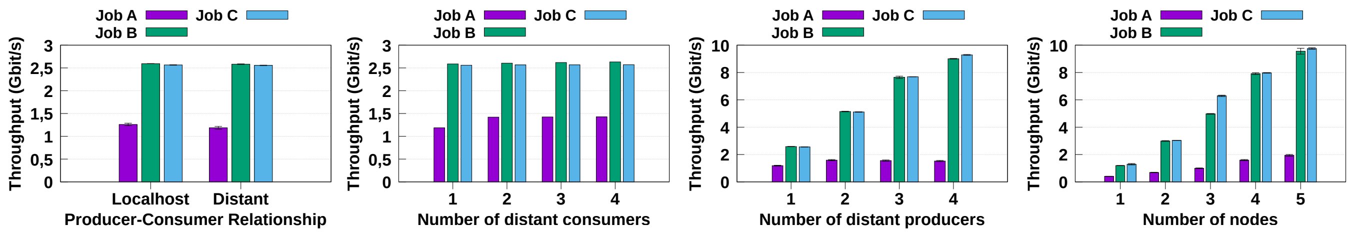
The most famous HPC oriented framework is MPI. Although we could have used MPI as an environment for running tasks and communicating data among them, many low-level optimizations that we performed would not have been possible or easily controllable (localhost communications through a UNIX socket, shared memory between processes to avoid copies, etc.). Moreover, it is difficult using MPI to assign specific tasks to specific nodes, to handle dynamic connection and disconnection, and to handle dynamic concurrent topology mappings. More recently, the StarPU versatile framework [18] proposes to create statically or dynamically [4] one or many tasks to perform computations, and can be coupled to MPI to run in a cluster. This framework is effective for implementing one-to-one and many-to-one topologies. But, compared to our proposal, it lacks a global view to execute concurrently different jobs on a single cluster, to implement efficiently many-to-many topologies, and to handle the output locations of several data streams.

In the last decade, plenty of big data frameworks were born. The most popular and active ones are certainly hosted by the Apache Software Foundation [19], and are gathering a large range of approaches. Mother of all, the underlying concept of *MapReduce* [20] is the central brain of Apache Hadoop [1]. It is able to process petabytes of data over a cluster, to perform data analytics for example. To be more efficient in problems with high data reuse, like machine learning, the Apache Spark [21] framework maximizes in-memory data storage and processing thanks to the concept



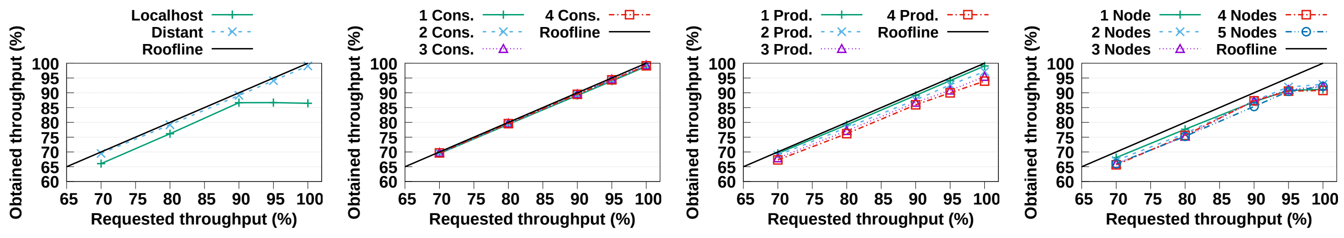
(a) one-to-one (b) one-to-many (c) many-to-one (d) many-to-many

Figure 4: Three different jobs running on different topologies: performance compared to the theoretical communication limits of the cluster.



(a) one-to-one (b) one-to-many (c) many-to-one (d) many-to-many

Figure 5: Three different jobs running on different topologies: performance of a real-world use case including computation.



(a) one-to-one (b) one-to-many (c) many-to-one (d) many-to-many

Figure 6: Measured throughput of job B on different topologies when the user reads the outputted data streams at constant speed (requested throughput). Throughputs are expressed as the percentage of the throughput obtained for the corresponding configuration in Figure 5.

of *Resilient Distributed Dataset* [22]. Apache Storm [23] is designed to be a distributed *event processing* framework able to process events coming from wide sources. Finally, the Apache Flink [2] framework proposes a flexible core able to perform any transformation on any stream. A common point in all of these big data frameworks is to be designed to process very large datasets over very large clusters with high-throughput, low latency considerations, and by handling frequent failures from the hardware or the software. Specially designed to process big data, these frameworks use distributed storage (distributed file system, distributed database, events and/or events streams) as inputs and as outputs. This approach is not efficient to meet our requirements of small input files and explicit output streams locations. Our framework targets concurrent processing of data streams with on-demand data delivery, that would not take advantage of these supported mechanisms.

Beyond these differences between existing frameworks and our proposal, some challenges are of course similar. One of these challenges is the resources allocation problem [18], [24], which is widely studied for HPC technologies like OpenCL and Cuda or for complex big data frameworks like the ones mentioned above. In the big data field these resource managers are called schedulers. The Hadoop project reflects their variety [25], providing various fine tuned scheduling policies. Some of them are noticeable in popularity (like YARN [26]) or innovative approach (like the Delay Scheduling [27]). These different resources management strategies can be used in our framework as customizable scheduling policies.

7. Conclusion

We presented a flexible and distributed runtime system architecture for generating constrained, high-throughput data streams. Its overall conception leads to a very adaptive and performance-oriented system for small to medium scale clusters, which is a typical context of such problems. We implemented this architecture in an experimental framework and evaluated its efficiency on a real-world professional digital printing use case mixing compute-intensiveness and high-throughput constraints. Our experimental study shows that our framework is able to nearly reach the maximal network bandwidth of the system in most of the tested scenarios, and provides excellent scalability.

Ongoing work target reactiveness and resilience improvements. We plan to improve the support of heterogeneous producers and make the scheduler able to duplicate tasks to ensure sustained throughput in case of a producer slowdown or crash.

Acknowledgments

We thank the Caldera company for supporting this work, providing the computer resources used in the experiments and the real-world use case to explore constrained data streams generation problems over different topologies.

References

[1] T. A. S. Foundation, “Apache hadoop,” https://hadoop.apache.org/, 2019, accessed: 2019-01-17.

[2] ——, “Apache flink,” flink.apache.org, accessed: 2019-01-10.

[3] ——, “Apache spark,” spark.apache.org, accessed: 2019-01-17.

[4] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, “Starpu-mpi: Task programming over clusters of machines enhanced with accelerators,” in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 298–299.

[5] M. Forum, “Message-passing interface (mpi) standard, version 3.1,” http://www.mpi-forum.org/, 2015.

[6] Z. Li and H. Shen, “Performance measurement on scale-up and scale-out hadoop with remote and local file systems,” in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 456–463.

[7] A. Toptal and I. Sabuncuoglu, “Distributed scheduling: a review of concepts and applications,” *International Journal of Production Research*, vol. 48, no. 18, pp. 5235–5262, 2010.

[8] R. Raman, M. Livny, and M. Solomon, “Matchmaking: Distributed resource management for high throughput computing,” in *hpdc*. IEEE, 1998, p. 140.

[9] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, “Toward exascale resilience,” *Int. J. of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[10] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[11] J. C. Sancho and D. J. Kerbyson, “Analysis of double buffering on two different multicore architectures: Quad-core opteron and the cell-be,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[12] J. Bhimani, N. Mi, M. Leeser, and Z. Yang, “Fim: Performance prediction for parallel computation in iterative data processing applications,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, June 2017, pp. 359–366.

[13] Y. Simmhan and L. Ramakrishnan, “Comparison of resource platform selection approaches for scientific workflows,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 445–450.

[14] M. Hovestadt, O. Kao, A. Keller, and A. Streit, “Scheduling in hpc resource management systems: Queuing vs. planning,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pp. 1–20.

[15] J. Nagle, “Congestion control in ip/tcp internetworks,” https://tools.ietf.org/html/rfc896, 1984, accessed: 2018-08-17.

[16] V. J. D. Borman, B. Braden and E. R. Scheffenegger, “Tcp extensions for high performance,” https://tools.ietf.org/html/rfc7323, 2014, accessed: 2018-08-17.

[17] L. A. Grieco and S. Mascolo, “Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control,” *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 25–38, 2004.

[18] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “Starpu: a unified platform for task scheduling on heterogeneous multicore architectures,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[19] T. A. S. Foundation, “Apache projects directory,” https://projects.apache.org/, 2019, accessed: 2019-01-10.

[20] J. Dean and S. Ghemawat, “System and method for efficient large-scale data processing,” Jan. 19 2010, uS Patent 7,650,331.

[21] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[23] T. A. S. Foundation, “Apache storm,” storm.apache.org, accessed: 2019-01-10.

[24] C. Gregg, J. Dorn, K. M. Hazelwood, and K. Skadron, “Fine-grained resource sharing for concurrent gpgpu kernels.” in *HotPar*, 2012.

[25] D. Yoo and K. M. Sim, “A comparative review of job scheduling for mapreduce,” in *Int. Conf. on Cloud Computing and Intelligence Systems (CCIS)*. Citeseer, 2011, pp. 353–358.

[26] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC ’13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[27] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.