HyperLoom: A Platform for Defining and Executing Scientific Pipelines in Distributed Environments

Vojtěch Cima
IT4Innovations, Czech Republic
vojtech.cima@vsb.cz

Stanislav Böhm
IT4Innovations, Czech Republic
stanislav.bohm@vsb.cz

Jan Martinovič
IT4Innovations, Czech Republic
jan.martinovic@vsb.cz

Jiří Dvorský
IT4Innovations, Czech Republic
jiri.dvorsky@vsb.cz

Kateřina Janurová
IT4Innovations, Czech Republic
katerina.janurova@vsb.cz

Tom Vander Aa
IMEC, Belgium
tom.vanderaa@imec.be

Thomas J. Ashby
IMEC, Belgium
ashby@imec.be

Vladimir Chupakhin
Computational Biology, Discovery
Sciences, Janssen Pharmaceutica NV
vchupakh@its.jnj.com

ABSTRACT

Real-world scientific applications often encompass end-to-end data processing pipelines composed of a large number of interconnected computational tasks of various granularity. We introduce HyperLoom, an open source platform for defining and executing such pipelines in distributed environments and providing a Python interface for defining tasks. HyperLoom is a self-contained system that does not use an external scheduler for the actual execution of the task. We have successfully employed HyperLoom for executing chemogenomics pipelines used in pharmaceutic industry for novel drug discovery.

KEYWORDS

HPC, Scientific Pipeline, Machine Learning, Big Data, Distributed Computing, Chemogenomics, Task Scheduling

1 INTRODUCTION

Scientific workloads are often composed of several consecutive computational phases. These phases are then combined into more complex data flows, which provide higher level functionality such as model cross-validation or hyper-parameter search. This results in *pipelines* having a shape of large directed acyclic computational graphs, whose nodes represent computational units – *tasks*. Figure 1 shows an example of such a pipeline. We present HyperLoom, an open source framework that simplifies definition and execution of end-to-end data processing pipelines in distributed environments. It is noteworthy that HyperLoom is a full-stack solution featuring its own task scheduling and execution engine that is not using any other resource scheduler as a backend.

Being aware that the area of executing tasks in distributed environments has been extensively studied for several decades, we describe several of the existing solutions in the context of our problem. Many of the existing and widely used data processing frameworks such as Hadoop [9], Spark [11], or HTCondor [5] do not allow fine grained inter-task dependencies to be specified. Tools such as SciLuigi [6], DAGman [2], or Pegasus [3] allow users to define custom inter-task dependencies but introduce other issues. These tools often rely on traditional resource schedulers that are optimized for coarse-grain long-running tasks and for which the time needed for resource allocation may create a significant scheduling overhead when executing short running tasks. Also, the inter-task data transfers are usually performed using a shared distributed file system, which becomes a performance bottleneck, especially in cases when large number of tasks generate a large number of I/O operations. Dask/Distributed [7] overcomes many of the limitations described above. Namely, it handles short running tasks and allows the filesystem usage to be reduced. However, similarly to the other tools, it does not support native pipelining of third-party applications.

HyperLoom is designed to mitigate the limitations mentioned above by implementing an optimized task scheduling algorithm, direct inter-task data transfer that reduces file system usage, a powerful task abstraction that enables to pipeline a variety of task types including third-party applications, scalable HPC native architecture, and a python API for easy user interaction. This is challenging for several reasons. The task execution time is not known in advance and may vary from milliseconds (short running tasks) up to days (long running tasks). Similarly, the size of the output generated by a task is not known before the task completes. Pipelines may contain a large number of various non-trivially interconnected tasks. Distributed environments, namely HPC clusters, contain thousands of computational cores, and different computational nodes may provide various resources with different capabilities.

This paper is organized as follows. Section 2 discusses HyperLoom design decisions and describes the architecture. Section 3 details HyperLoom task scheduling process. We evaluate and discuss HyperLoom performance in Section 4. Finally, we conclude in Section 5.

PARMA-DITAM '18, Manchester, United Kingdom
© 2018 ACM. 978-1-4503-6444-7/18/01…$15.00
DOI: 10.1145/3183767.3183768

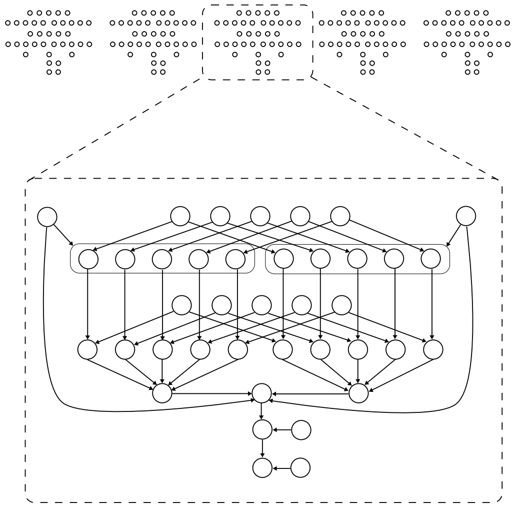

Figure 1: An example of a HyperLoom pipeline visualized as a directed acyclic graph where graph nodes represent computational tasks.

2 ARCHITECTURE

This section discusses the HyperLoom design philosophy and architecture based on the challenges described above.

2.1 Design Decisions

To design a solution that tackles the challenges introduced earlier, we define the following design goals.

Low scheduling overhead – The scheduling process has to be sufficiently fast since we also have to deal with relatively short running tasks (< 1 second) for which the scheduling time may represent a significant portion of the actual execution time. **Sufficient scheduling quality** – The scheduler should plan tasks on a cluster while utilizing as many resources as possible, while desirably consuming the minimal amount of the resources by the scheduling process itself. Since we do not know computational characteristic of tasks in advance, we do not aim to compute the best optimal task placement. Even if we had all the information, obtaining the optimal solution is computationally unfeasible since the scheduling problem is generally NP-hard. Therefore, our goal is to design a heuristic that achieve good results in practice considering industry driven use-cases. **Extensibility** – We aim to provide a generic platform that allows an easy integration and chaining of existing tools including black-box applications (e.g. third party applications) with a possibility to specify their resource requirements as well as to define arbitrary task types directly. **Portability** – Our intention is to build a generic open-source framework that can be adopted and used by a large variety of research groups using different types of distributed systems ranging from HPC to Cloud environments.

2.2 Overview

Figure 2 shows the main components of HyperLoom. The components can be categorized into frontend and backend sections. The computational backend of HyperLoom consists of *worker* components managed by a centralized *server*. Worker processes operate on computational nodes and execute tasks as scheduled by the server. Server reactively schedules tasks respecting task resource requirements and resources available on workers at the time of scheduling. The frontend of HyperLoom only contains a lightweight *client* component (Python3 module) that allows tasks to be defined and chained into pipelines as well as the pipelines to be submitted to the server. It also allows the results to be gathered back once the computation completes.

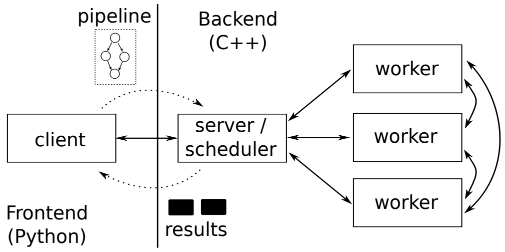

Figure 2: HyperLoom architecture.

We highlight the following design features of HyperLoom: **In-memory data storage** – By default, data produced by a task is held in the worker RAM memory (RAMDisk) when needed for further use by other tasks. **Reactive scheduling** – The scheduler processes tasks reactively as the computation proceeds. One of the main objectives of the scheduler is to reduce inter-worker data transfer by moving computation to data. HyperLoom scheduler is discussed in detail in Section 3. **Direct worker-to-worker communication** – Although scheduler aims to reduce inter-worker data transfers, sometimes they are necessary to utilize the cluster efficiently. Therefore HyperLoom allows that data produced by a task on a worker can be fetched from any other worker directly with no server or file system overhead. **Powerful task abstraction** – HyperLoom offers a predefined set of task types. These types cover tasks on a level of simple constants, file operations, Python tasks, or binaries. All of the types can be employed using the Python API. **Performance visualization** – HyperLoom includes a tool for providing insights on pipeline performance from various angles. For example, task execution overviews, scheduling details, or utilization of worker resources, which helps to identify and debug possible performance bottlenecks in the pipeline.

3 SCHEDULING

Task placement is a crucial property which has a significant impact on the overall performance. In order to achieve the design goals defined in Section 2.1 while considering the problem properties introduced in Section 1, we have made the following design choices:

Reactive scheduling – Unknown and imbalanced task characteristics makes it impossible to statically divide a part of the pipeline to each worker and expect an efficient and balanced execution. Therefore, the scheduling process needs to be a reactive process respecting the current load on a cluster in real time as the pipeline execution proceeds.

Different strategies depending on the number of pending tasks – For the scheduler, it is a very different situation when there are only a few enabled tasks and the cluster is not fully utilized, or when there are hundred thousands of enabled tasks, the scheduler is overloaded and there are many options how to schedule tasks to workers.

We want to reflect this distinction in the scheduler. When there are many enabled tasks, tasks can be assigned to a worker even though the worker resource capacity may not be sufficient for the task at the time of assignment. This allows to overlap the communication by computation with a low risk of starving due to the improper assignment. Although each worker maintains a set of tasks assigned to it, which may in total exceed the worker resource capacity, it only executes the task when the required capacity becomes available, i.e. workers do not actually overbook their resources. Moreover, since having many enabled tasks usually implies many intermediate resources, the scheduler tries to choose a strategy that leads to the decrease in the number of enabled tasks to free memory.

Data locality and replication – Data created by a task may generally have a significant size; therefore, the scheduler utilizes a certain level of data locality to avoid unnecessary data transfers. In situations where, from various reasons, some data has to be transfered to several workers, every of those workers keeps the data available in-memory as independent replicas for further computations. The fact that, by default, all the data needed for further computation are always kept in the workers RAM memory and can be possibly replicated over more workers introduces a very little to no overhead to cluster file system.

3.1 Definitions

Let *pipeline* be a tuple (T, I, C) where T is a finite set of tasks and I is a mapping $I : T \longrightarrow 2^T$ where $I(t)$ is a set of input tasks that has to be completed before t can be executed. $C : T \longrightarrow \mathbb{N}$ is a number of cores needed for the execution of the task. Let us note that in HyperLoom, inputs of each task are ordered; however, for the purpose of defining the scheduling algorithm, ordering is not important and we use only sets of input tasks. Also C can be simply generalized to describe more than one resource; however, for the sake of simplicity, we stay with only one resource in the description. The mapping $O : T \longrightarrow 2^T$ is defined as $O(t) = \{t' \in T \mid t \in I(t')\}$ and represents the set of output tasks. In the example in Figure 3, we have $T = \{1, \ldots, 9\}$ and $I(7) = \{4, 6\}$, $O(7) = \{8, 9\}$, $C(7) = 4$.

For the rest of the text, we fix a pipeline (T, I, C) and a finite set of workers W together with a function $R : W \longrightarrow \mathbb{N}$ that defines the number of cores in each worker. In the example, $W = \{\mathbf{A}, \mathbf{B}\}$, $R(\mathbf{A}) = 4$, and $R(\mathbf{B}) = 6$.

Now, we define mappings S and P to describe a state of a pipeline execution: $S : T \longrightarrow \{\bot\} \cup \mathbb{N}$ (size of task results) and $P : T \longrightarrow 2^W$ (task placements). If $S(t) = \bot$ than t has not been computed yet and its size is unknown; if $S(t) \in \mathbb{N}$ then t was computed and the $S(t)$ is the size of the resulting data. Mapping P assigns tasks to workers. If $S(t) \neq \bot$ then $P(t)$ determines which workers hold the result of t. In the case of $S(t) = \bot$ then if $P(t) = \{w\}$ means that t is currently computed on a worker w; otherwise $P(t)$ has to be the empty set (t was not assigned to any worker). Some values for S and P in our example are $S(1) = 2048$, $S(5) = S(7) = \bot$, $P(1) = \{\mathbf{A}\}$, $P(7) = \{\mathbf{B}\}$, and $P(5) = \emptyset$.

Now we define the following two sets and a function related to S and P:

- Finished tasks: $\mathcal{F}_S = \{t \in T \mid S(t) \neq \bot\}$
- Pending tasks: $\mathcal{P}_{S,P} = \{t \in T \mid I(t) \subseteq \mathcal{F}_S \wedge P(t) = \emptyset\}$
- Sum of the data placed on worker w for a set of tasks X:

$$D_{S,P} : 2^T \times W \longrightarrow \mathbb{N}$$

where

$$D_{S,P}(X, w) = \sum_{t \in X \,\wedge\, w \in P(t) \,\wedge\, S(t) \neq \bot} S(t)\,.$$

Finally, we define the helping function Bound

$$\text{Bound}_n(X) = X \text{ if } |X| \leq n \text{ otherwise } \emptyset\,.$$

In the following text, we use constants that represent parameterization of the heuristic algorithm; all of them are denoted by symbol Ψ and they are defined as they appear.

3.2 Scheduling Score Function

The scheduling algorithm (Algorithm 1) is implemented as an iterative process that assigns *one* task *each round* until there are no pending tasks or free resources on workers. In each iteration, a "score" value is computed for each pair of a task t and a worker w if there are enough resources to run t on w. The value expresses how efficient is to plan the task t to the worker w. If a pair (t, w) has the highest score than the task t is assigned to worker w, and the process is repeated. In the following iteration, the score values may be changed since assigning a task to a worker may change the data placement.

Note that the score value serves for two purposes at once: (1) to choose the task among all pending tasks for the current assignment and (2) to choose the most suitable worker for the selected task.

Algorithm 1 Server scheduling loop

```
Fill a set of pending tasks ($\mathcal{P}$) by initial tasks (tasks with no inputs)
while $\mathcal{P} \neq \emptyset$ or there is a running task do
    Schedule tasks from $\mathcal{P}$ on workers (up to resources of workers)*.
    Remove scheduled tasks from $\mathcal{P}$
    Wait for a message that a task is finished
    while Process all pending message do
        Receive a message that a task $t$ was finished.
        Add new tasks to $\mathcal{P}$ that was enabled by finishing $t$.
    end while
end while
```

The $\text{Score}_{S,P}$ function is defined as follows

$$\text{Score}_{S,P}(t, w) = \text{Score}'_{S,P}(t, w) + \text{Score}''_{S,P}(t, w) + \text{Score}'''_{S,P}(t, w)\,,$$

where

- $\text{Score}'_{S,P}$ represents a proximity of directly interconnected tasks,
- $\text{Score}''_{S,P}$ represents task resource requirements,

- $\text{Score}'''_{S,P}$ represents a proximity of tasks with a common successor.

We discuss the contribution of each of those components in the following subsections.

3.2.1 Proximity of directly interconnected tasks. It is desired to run a task at a worker that already contains the data required for the task execution. Therefore, the scheduler favors a placement inducing the lowest possible data transfer at a time.

The following scenarios describe two of basic properties that we want from score for tasks t_1 and t_2.

Scenario: One has everything, other nothing Let assume that all data for task t_1 are located on the worker w_1 and other workers have nothing; the size of input data for t_1 on worker w_1 is 1 GB. Similarly, all data for task t_2 are located on the worker w_1 and other workers have nothing; the size of input data for t_2 on worker w_1 is 2 GB. Obviously, we would like to achieve that the score for (t_1, w_1) is a large positive number and score for all $(t_1, w), w \in W \setminus \{w_1\}$ should be a (relatively) large negative value. This also holds for the case of t_2. Because of the data sizes, assigning t_2 to w_1 is more important than t_1 to w_1. Moreover, if we have to compute one of these tasks on a worker different from w_1, it is better to choose t_1. The score function has to respect this.

Scenario: Equality Assume that executing t_1 on any worker needs to transfer same amount of data. For instance, (Example A) all workers have all data (zero transfer for all) or (Example B) each worker has a unique piece of the data with the same size that others do not have. In such case we want to assign zero score for each pair containing t_1 since there is no affinity to any worker even we have to transfer some data (Example B).

The score value is a dimensionless quantity; but it can be very roughly interpreted as follows: If score for pair (t, w) is a positive number s, then we can expect to transfer s bytes more (in average) when t is assigned to a different worker than w. If s is a negative number, then we could transfer $-s$ bytes less (in average) if we assign data to different than w.

$\text{Score}'_{S,P}$ is computed as follows

$$\text{Score}'_{S,P}(t, w) = D_{S,P}(I(t), w) - \frac{\sum_{w' \in W} D_{S,P}(I(t), w')}{|W|}.$$

3.2.2 Resource requirements. Score$''$ favors tasks with higher resource requirements as these are more challenging to be scheduled (bonus for each extra cpu) and also favors tasks with more successors (to prioritize tasks that may enable other tasks that can be processed in parallel).

$$\text{Score}''_{S,P}(t, w) = \max\{C(t) - 1, 0\}\, \Psi_{cpu} + |O(t)| \Psi_{output},$$

where constants Ψ_{cpu} and Ψ_{output} control the level of contribution of both members.

3.2.3 Proximity of tasks with common successor. Let us consider a set of tasks with a common successor. Given the fact that such successor requires outputs of all its direct ancestors in order to run, it is desired to schedule the tasks to run close to each other (ideally at the same worker). When tasks are not executed at the same worker, an inter-worker data transfer is induced.

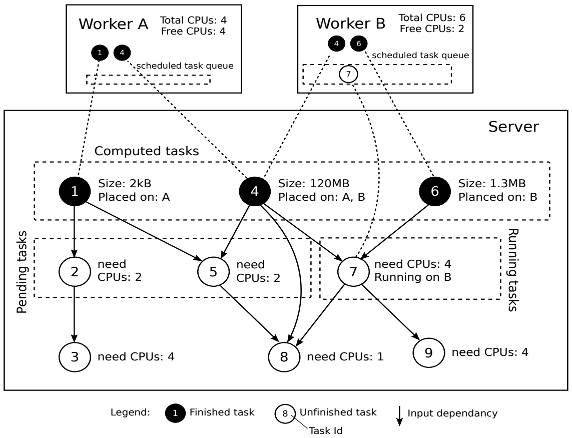

Figure 3: An example of a pipeline execution state

Score$'''$ considers data locality of the inputs of tasks as follows

$$\text{Score}'''_{S,P}(t, w) = \sum_{t' \in \text{Bound}_{\Psi_{ex}}(O(t))} \min\left\{\Psi_{limit}, \frac{D(\text{Bound}_{\Psi_{ex}}(I(t')) \cap \mathcal{F}_S, w)}{\Psi_{factor}}\right\},$$

where Ψ_{ex}, Ψ_{limit}, and Ψ_{factor} are tunable constants.

Generally, Score$'''$ is restricted and bounded for two reasons. First, the situation in the cluster may be a very different when t is actually finished; therefore, the impact of sizes of "neighbor tasks" is only approximated. Second, a task may have a large number of neighbors, hence Score$'''$ is computed only if there is a relatively small number of neighbors for performance reasons. Usually, when there are many neighbors, they are spread out all over the nodes so the impact of Score$'''$ on choosing the best worker for t is limited.

3.3 Scheduling Algorithm

The presented architecture allows to dynamically react on a given situation and utilize real-time information: what is the current utilization of workers and real sizes of the results produced by finished tasks (i.e. we can estimate the cost of data transfers).

Figure 3 captures a pipeline execution in the following state:

- Tasks 1, 4, and 6 have finished. Task 1 has been finished right now; this event puts tasks 2 and 5 into the set of pending tasks.
- Task 7 is running.
- Inputs for task 3, 8, and 9 have not been computed yet and thus cannot be executed.

In this situation, we can expect that the scheduler dispatches task 2 on worker A, and task 5 on worker B. This also triggers a data transfer of the result of task 1 from worker A to worker B.

We describe the scheduling algorithm more formally in Algorithm 2.

Algorithm 2 Scheduling algorithm

```
function ScheduleTasks($S, P$)
    $S_1 \leftarrow S$
    $P_1 \leftarrow P$
    $R_1 \leftarrow R$
    $\ell \leftarrow \max\{\Psi_{mintasks}, \Psi_{freecpu} \sum_{w \in W} R(w)\}$
    $T' \leftarrow$ Subset of $\mathcal{P}_{S,P}$ with at most $\ell$ elements with minimal ids.
    for $i = 1, 2, \ldots$ do
        $X \leftarrow \{(t, w, x) \in T' \times W \times \mathbb{N} \mid C(t) \leq R_i(w) \wedge s = \text{Score}_{S_i,P_i}(t, w)\}$
        if $X = \emptyset$ then
            return($S_i, P_i$)                ▷ Return a new $S$ and $P$
        end if
        $(t_R, w_R) \leftarrow (t, w)$ such that $(t, w, s) \in X$ and $s$ is minimal (among $X$).
        Assign task $t_R$ to worker $w_R$.
        $S_{i+1} = \lambda t.$ if $t = t_R$ then $\Psi_{size}$ else $S_i(t)$
        $P_{i+1} = \lambda t.$ if $t = t_R$ then $P_i(t) \cup \{w_R\}$ else $P_i(t)$
        $R_{i+1} = \lambda w.$ if $w = w_R$ then $R_i(w) - C(t)$ else $R_i(t)$
    end for
end function
```

4 PERFORMANCE

We evaluated HyperLoom performance and scalability through a series of experiments on a physical testbed.

We evaluate the performance by measuring overall pipeline execution time – the period of time it takes for the pipelines to be completed. We measure the elapsed time between the submission and successful completion of the pipeline.

4.1 Test Scenarios

We have designed three test cases. Two synthetic, devoted to evaluate performance of HyperLoom in comparison to Dask/Distributed, and one derived from a compound-activity modeling pipeline to demonstrate the scalability of HyperLoom for real-world application.

50kh – a synthetic test case designed to generate intensive scheduling load. The assembled pipeline contains 50k independent tasks that each executes the *hostname* program. Since this program completes instantly, it forces the scheduler to react promptly in order to keep workers utilized.

gridcat – a synthetic test case designed to evaluate scheduling quality. The assembled pipeline contains 40 tasks that each generate 200MB of data, followed by a layer of 1,600 tasks that represent concatenations of every possible pair of the data generated in the first layer, followed by a layer of another 1,600 tasks that compute *md5* hashes of the concatenated data. If the scheduler does not utilize the location of the data, it will induce a significant inter-worker data transfer.

mlchemo – a test case derived from an existing scientific workflow used for novel drug discovery. This pipeline performs a nested 5×5 cross-validation with hyper-parameter search for machine-learning based models capturing compound-activity prediction. The shape of this pipeline is similar to the one depicted in Figure 1. The pipeline contains a mix of long running tasks such as modeling and validation done by LibSVM [1] – a widely used support vector machine implementation and short running tasks providing auxiliary functionality.

Table 1: Comparison of the pipeline execution time [s] in HyperLoom and Dask/Distributed (*50kh, gridcat*)

	50kh		*gridcat*	
# workers	HyperLoom	D/D	HyperLoom	D/D
1	141.48	359.00	119.78	N/A
8	19.66	81.91	40.47	N/A
16	11.24	71.03	47.72	360.72
32	17.41	73.10	43.42	162.00
64	34.28	73.80	41.98	89.45

4.2 Testbed

All the experiments have been performed on a dedicated testbed using up to 64 identical physical computational nodes, each with two 12-core Intel Xeon E5-2680v3 processors (2.5GHz)[1] and 128 GB of physical RAM memory. The nodes are interconnected by 7D Enhanced hypercube Infiniband [10] network (56 Gbps). Nodes run Red Hat Enterprise Linux 6.5 [4].

4.3 Experiment 1: Scheduling Overhead

We contrast the scheduling overhead in HyperLoom and Dask/Distributed by comparing the pipeline execution time of the *50kh* test case that contains large number of independent short running tasks. Thus, *50kh* is expected to stress the reactive scheduling process which allows us to analyze the scheduling overhead.

Table 1 compares execution time of *50kh* using both, HyperLoom and Dask/Distributed, executing the pipeline on 1, 8, 16, 32, and 64 workers. In all of the cases, HyperLoom significantly outperforms Dask/Distributed completing the pipeline in less than half of the time. As the pipeline only contains independent short running tasks, we argue that the performance difference in this case is caused by the higher scheduling overhead.

4.4 Experiment 2: Scheduling Quality

In some cases, tasks generate significant amount of data. As a consequence, a suboptimal task placement results in delays due to data transfer between workers. In this regard, we have designed the *gridcat* test case to simulate this type of scenarios. We measure the execution time of *gridcat* in both, HyperLoom and Dask/Distributed using 1, 8, 16, 32, and 64 computational nodes. While HyperLoom successfully completes the pipeline execution in all of the test scenarios, the Dask/Distributed implementation fails when using less than 16 nodes due to an out-of-memory error. In all of the cases when both of the implementations finish, HyperLoom significantly outperforms Dask/Distributed. The exact figures for this experiment can be found in Table 1.

[1] $http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz$

Table 2: HyperLoom scalability - strong and weak scaling experiments performed using the *mlchemo* test case.

	Strong Scaling		Weak Scaling	
# workers	t [s]	SSE	t [s]	WSE
1	29,363	1.00	334	1.00
8	3,576	1.03	338	0.99
16	1,817	1.01	351	0.95
32	1,020	0.90	368	0.91
64	559	0.78	374	0.89

4.5 Experiment 3: Scalability

We evaluate HyperLoom scalability using the *mlchemo* test scenario executing the pipeline on 1, 8, 16 and 64 workers (24 CPU cores each) and measure the total execution time for each. We demonstrate the performance for both, weak and strong scaling.

For the strong scaling experiments, we only increase the number of workers while keeping the pipeline size constant (~460k tasks). For the weak scaling experiments, we linearly increase the pipeline size with the increasing number of workers by replicating a base *mlchemo* pipeline (~12.5k tasks × # workers).

We compute strong scaling efficiency (SSE) as follows

$$SSE = \frac{t_1}{Nt_n}\,, \tag{1}$$

where t_1 is the execution time running on a single worker, N is the number of workers and t_n is the execution time running on N workers.

We compute weak scaling efficiency (WSE) as follows

$$WSE = \frac{t_1}{t_n}\,, \tag{2}$$

where t_1 is the execution time running on one worker, and t_n is the execution time running on N workers.

In Table 2, we observe almost linear decrease of the execution time with the increasing number of workers. Concretely, the execution time decreases from more than 8 hours (1 worker) to less than 10 minutes (64 workers). The SSE values are derived from the respective execution times using equation 1. Although, in the long run, the SSE decreases with the increasing number of workers, the observed decrease is very moderate; from SSE 1.0 (1 worker) to SSE 0.8 (64 workers). It is noteworthy that for the cases with 8 and 16 workers we even observe SSE to be slightly higher than 1.

Table 2 also shows the execution time for the weak scaling experiments where the number of tasks employed in the pipeline increases linearly with the number of workers. Ideally, we would expect that the execution time remains constant for all of the experiments. Nevertheless, increasing the pipeline and cluster size also increases the overall system overhead, which causes the execution time to increase. In particular, the execution time increases from 334 seconds (1 worker, ~12.5k tasks) to 374 seconds (64 workers, ~800k tasks). The WSE values are derived from the respective execution times using equation 2. WSE slightly decreases from 1 (1 worker) to 0.9 (64 workers).

5 CONCLUSIONS

We introduced HyperLoom, an open-source platform for an efficient definition and execution of scientific pipelines in distributed environments. HyperLoom enables to chain large number of computational tasks into a complex end-to-end data processing pipelines using a simple Python interface as a gateway to the high-performance backend of HyperLoom.

We analyzed HyperLoom performance using both synthetic and real test cases scaling up to hundreds of thousands of tasks distributed across hundreds of CPU cores. HyperLoom significantly outperformed Dask/Distributed in synthetic test cases ranging from ~6.3× to ~2.2× better performance for the *50kh* test case and from ~7.6× to ~2.1× better for the *gridcat* test case. We have also successfully deployed a pipeline to address the challenge of generating compound-target activity predictions for publicly available big chemogenomics datasets [8], which proves HyperLoom potential to be used for real-world end-to-end data processing applications.

6 ACKNOWLEDGEMENTS

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671555 (ExCAPE) and by the IT4Innovations infrastructure which is supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project "IT4Innovations National Supercomputing Center – LM2015070".

REFERENCES

[1] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)* 2, 3 (2011), 27.

[2] Weiwei Chen and Ewa Deelman. 2011. Workflow Overhead Analysis and Optimizations. In *Proceedings of the 6th Workshop on Workflows in Support of Large-scale Science (WORKS '11)*. ACM, New York, NY, USA, 11–20. DOI: http://dx.doi.org/10.1145/2110497.2110500

[3] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. 2005. Pegasus: A Framework for Mapping Complex Scientific Workflows Onto Distributed Systems. *Sci. Program.* 13, 3 (July 2005), 219–237. DOI: http://dx.doi.org/10.1155/2005/128026

[4] Red Hat. 2017. Red Hat Enterprise Linux. (2017). https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux [Online; accessed 31-March-2017].

[5] HTCondor. 2017. HTCondor. (2017). https://research.cs.wisc.edu/htcondor/index.html [Online; accessed 31-March-2017].

[6] Samuel Lampa, Jonathan Alvarsson, and Ola Spjuth. 2016. Towards agile large-scale predictive modelling in drug discovery with flow-based programming design principles. *Journal of Cheminformatics* 8, 1 (2016), 67. DOI: http://dx.doi.org/10.1186/s13321-016-0179-6

[7] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*. Citeseer, 130–136.

[8] Jiangming Sun, Nina Jeliazkova, Vladimir Chupakin, Jose-Felipe Golib-Dzib, Ola Engkvist, Lars Carlsson, Jörg Wegner, Hugo Ceulemans, Ivan Georgiev, Vedrin Jeliazkov, Nikolay Kochev, Thomas J. Ashby, and Hongming Chen. 2017. ExCAPE-DB: an integrated large scale dataset facilitating Big Data analysis in chemogenomics. *Journal of Cheminformatics* 9, 1 (dec 2017), 17. DOI: http://dx.doi.org/10.1186/s13321-017-0203-5

[9] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[10] Wikipedia. 2017. InfiniBand — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=InfiniBand&oldid=772443735 [Online; accessed 31-March-2017].

[11] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, and others. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.