Project no. 801091

ASPIDE

Research & Innovation Action (RIA)
EXASCALE PROGRAMING MODELS FOR EXTREME DATA PROCESSING

Data collection and mining for Exascale systems
D2.3

Due date of deliverable: 14th December 2019

Start date of project: June 15^{th}, 2018

Type: Deliverable
WP number: WP2

Responsible institution: University Carlos III of Madrid
Editor and editor's address: Javier Garcia-Blas, University Carlos III of Madrid

Version 2.0

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision History

Rev.	Date	Author(s)	Description
1.0	31/10/19	Javier Garcia-Blas (UC3M)	Initial Version.
1.1	26/11/19	David del Rio Astorga (UC3M)	Added Chapter 5.
1.2	09/12/19	Lionel Vincent (ATOS/BULL)	Add Bull IO Instrumentation metrics in Chapter 2
1.3	10/12/19	David del Rio Astorga (UC3M)	Extended interface definitions in Chapter 5.
1.4	10/12/19	Javier Garcia-Blas (UC3M)	Added Chapter 3.
1.5	10/12/19	Gabriel Iuhasz (IeAT)	Added details on work related to data locality anomalies an metrics in Chapter 2
1.6	10/12/19	Gabriel Iuhasz (IeAT)	Added details details about hardware and software anomalies in Chapter 2
1.7	13/12/19	Adrian Spătaru (IeAT)	Added content in Chapter 4.
1.8	13/12/19	Javier Garcia-Blas (UC3M)	Modified the structure of the document.
1.9	14/12/19	Javier Garcia-Blas (UC3M)	Added introduction chapter.
2.0	14/12/19	Javier Garcia-Blas (UC3M)	Minor issues and typos.

Main Contributors

Adrian Spătaru (IeAT)
David del Rio Astorga (UC3M)
Gabriel Iuhasz (IeAT)
Javier Garcia-Blas (UC3M)

Lionel Vincent (ATOS/BULL)

Executive Summary

This deliverable covers the description of the explored data collection and mining methodologies for improving data placement in data-intensive applications.

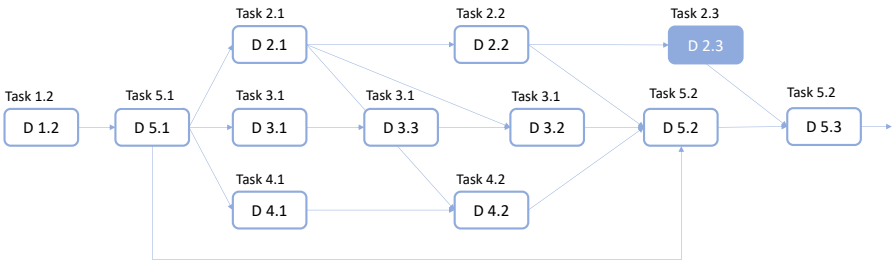

Figure 1: Deliverable dependencies of D2.3.

Contents

Chapter 1

Introduction

As data intensive scientific computing systems become more widespread, there is a necessity of simplifying the development, deployment, and execution of complex data analysis applications for scientific discovery. The scientific workflow model is the leading approach for designing and executing data-intensive applications in high-performance computing infrastructures.

Current scheduling schemes are data locality agnostic. In this mode, tasks are first placed in the work queue of the node where they are created. If no work stealing events are triggered, each task remains there until a worker attached to the server obtains the task. If another server steals the task, it can be executed on any worker in the system. Thus, this scheme is not predictable, but it allows total flexibility to the load balancer. However, the application of this scheme in a data-intensive use case would result in a great increase of network traffic since locality is not applied.

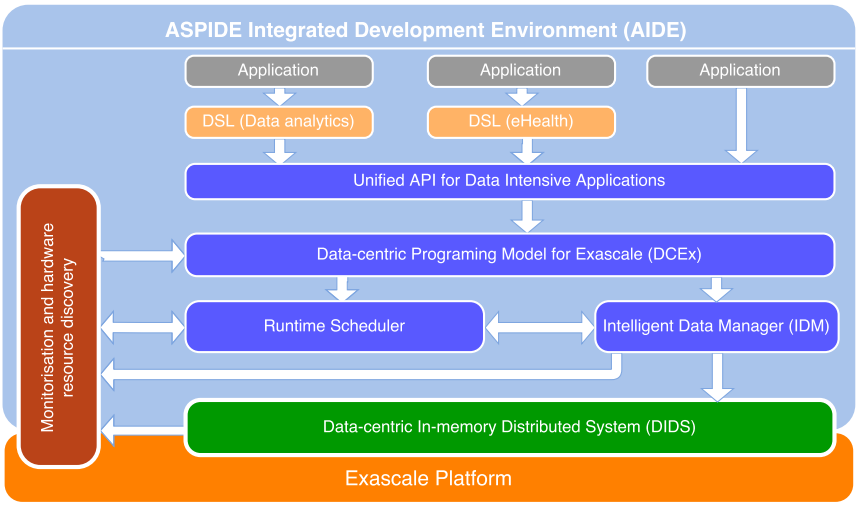

Figure 1.1: ASPIDE architectural design.

The data placement mechanisms proposed in **ASPIDE** and integrated in the **IDM** component are key elements for enhancing data-locality as shown in Figure 1.1. Data placement has to be complemented with task placement management in order to open up an optimization space that can be used for trading off load balance and data locality.

The aforementioned data placement strategies included in the IDM component can be improved with novel techniques for adaptive data and task placement by taking into consideration dynamic load conditions and capacities of the system. There are a lot of metrics that can be considered: current free space in I/O nodes, current computing load, current I/O load, or even the characteristics of the model (place different inputs of one specific task in the same I/O node). The monitoring service provides feedback to the runtime scheduler with the objective of auto-tune applications in a dynamic fashion. Finally, this service provides a mechanism for detecting anomalies in the in-memory data-aware subsystem, for example inefficiencies in the network or memory capacity. It is important to remark that our system cohabits with running applications, so an efficient memory management must be covered as well. This unprecedented large volume of data brings new constraints in terms of discovery, access, exploitation, and visualization.

The remainder of the deliverable is organized as follows. First, Chapter 2 introduces the metrics employed for enhance data locality in data-intensive applications. This chapter covers the identified anomalies and events that can impact in data locality. This chapter also describes the instrumentation carried out for profiling I/O in Exascale system, with the aim of studying the impact of multiple IO metrics such as the number of IO operations, cache misses, the IO duration, etc. In Chapter 3, we detail the process of matchmaking tasks, data, and computational nodes. Later, Chapter 4 presents the high-level interfaces for creating tasks and task pools to be scheduled and executed by the runtime. We also describe in which way the aforementioned metric collection is employed by the schedule to assist the task execution. In Chapter 5, we describe how our proposed AIDE-based solution has been integrated with the proposed programming model for efficiently supporting many-task workflow applications and how it maps with the special requirements of data-intensive applications. Lastly, Chapter 6 concludes the presented work.

Chapter 2

Metrics for enhance data locality

2.1 Anomalies and Events effect on data locality

When we talk about Exascale systems we usually think of the actual size and variety of hardware as well as software stacks necessary for the operation of such systems. Some technologies and fields of computer science have the characteristics of Exascale however, none have all of them. Data locality for example is also one of the key features for Big Data systems such as Hadoop and Spark.

In these systems computation is moved close to were the data is actually stored, not the other way around. Any computation requested by a data intensive application is much more efficient if it is executed near the data it operates on. We can easily see that this is more so when the data we are dealing with is of substantial size. This way of doing things minimizes network traffic which in turn increases system throughput. HDFS[1] has mechanisms and interfaces that allow applications to move themselves closer to where the data is located.

In order to take advantage of data locality, we need to satisfy certain conditions. First, cluster have to have an appropriate topology. The application must have the ability to read data locally. The framework also has to know the topology of the nodes on which tasks are executed and know the location of data to be processed.

For example in Hadoop we have the concept of "Racks[2]". Using this concepts we can define three types of data locality:

- *Local Data*: In this case the data is located on the same node as the mapper working on the data. This is the optimal scenario.
- *Intra-Rack data locality*: Mapper cannot be executed on the same node as

[1] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[2] https://hadoop.apache.org/docs/r1.2.1/cluster_setup.html

the date due to resource constraints. In this scenario the optimal available solution would be to run the mapper on a different nearby node, but located in the same "Rack".

- *Inter-Rack data locality*: This is the worst case scenario, where no node situated in the same Rack as the data is available to run the mapper.

In theory, this works rather well. In practice there are many more issues to contend with. In the case of Exascale systems in particular the efficiency of computation and data locality can be effected also by speculative execution, heterogeneous computational resources or event events such as hardware faults or any other performance related anomaly.

In essence, this strategy tries to remove wherever possible the bottleneck created by moving large quantities of data between nodes. It is important that we don't consider only computational time as a bottleneck but also I/O inside the node and outside the node. Most data intensive applications try to optimize their data placement for production system so that it heavily favors local data. Usually around 80-90% of I/O is local, 5-10% is Rack local and 2-5% is remote. This rescheduling is also heavily impacted by potential faults in the underlying hardware and software stacks. In the following section we will discuss and define some of these anomalies and events.

2.1.1 Events and Anomalies

As we have already described in Deliverable D3.2 [1] and D3.3 [2] the size of Exascale systems guarantees that at any given time there will be an event or anomaly which impacts performance, typically every 30 minutes [3]. Also in those deliverable we described both the metrics and the Event Detection Engine (EDE). In the following sections we will describe and characterize some of the events and anomalies which are present in large scale distributes systems and we feel are pertinent for Exascale.

We define an event which impacts system performance and stability as an anomalous behaviour at the hardware or software level that can lead to invalid systems states or errors as well as in the worst case scenario operation interruptions [4]. It is beyond the scope of this deliverable to detail the methodologies used to detect these events and anomalies. We will instead focus on the anomalies themselves. Examples of methods for detection can be found in Deliverable D3.3 [2].

A lot of research is being done for detecting hardware failures and the anomalies they induce [4–6]. As access to metrics and logs from real world production systems is nigh on impossible to find a lot of available datasets are also synthetically generated. Moreover, even the generated datasets rarely focus on distributed systems, instead representing one node workloads. Analyzing monitoring data from a distributed

system as a whole could yield a much better understanding of anomalous events.

Some hardware anomalies include but are not limited to [5, 6]:

- *leak* - Symbolizes a memory leak, causing memory fragmentation and usually severe system slowdown.
- *memeater* - A memory interference fault and saturating bandwidth.
- *ddot* - CPU and cache interference fault, resulting in degraded performance for affected CPUs.
- *dial* - ALU interference fault, resulting in degraded performance for applications running on the same Core as where the fault occurs.
- *cpufreq* - Falling CPU frequency usually caused by misconfiguration or failure in cooling and/or CPU fault.
- *pagefail* - Failure in page allocation requests caused by misconfiguration or system memory failure. Can cause failing or stalling applications.
- *ioerr* - Caused by failing HDD I/O operations indicating HDD imminent failure.
- *copy* - Caused by I/O interference or failing HDD. Usually presents as a saturation or plateauing of the I/O bandwidth Badly effects I/O bound applications.
- *linklog* - Delay before MPI messages are sent. Usually caused by network connectivity or congestion issues.

The anomalies listed here where used for the experiments detailed in Deliverable D3.3, Section 6.2.

Hardware faults

The above enumerated anomalies are interesting as they can be (and are) synthetically generated by using custom scripts or applications [5]. Almost all of these mimic hardware failures. These can, and do appear in any part of an Exascale system. Some faults can be somewhat attenuated by redundant systems (i.e. backup batteries) while others can cause parts of an application to be rescheduled or even complete system halt.

There are generally three types of hardware failures [3]. The first type are those errors which are detected and corrected by hardware. One example of these ECC RAM modules. Even if these types of faults are detectable by hardware and are also resolved by hardware supervisory software systems are used to collect data regarding these for later error analysis. In **ASPIDE**, this is the role of the monitoring and EDE systems.

The second type of hardware fault are those faults which although detected by hardware cannot be fixed by hardware (i.e. double bit error). These faults, same as before, are logged and collected by the monitoring system. These faults can usually result in application restart, application halt or compute node restart.

The final type of faults are the ones which are the most challenging to detect and classify. These fault are silent from the point of view of the hardware. They need to be detected using external software tools such as EDE. These errors include; incorrect memory and network transfer, control errors which cause premature termination of a task or computation.

Some of the aforementioned errors are easier to detect than others. Different components of an Exascale system fail at different rates. Some of the causes of failure can be modeled statistically while others are the result of transient conditions. In our opinion the most important types of hardware faults to detect and run root cause analysis on are the silent hardware faults. Once these where successfully detected the resulting data can be used for adaptive self-tuning and optimization mechanisms which will aid in reducing overall system performance degradation and data locality.

Software faults

In the case of software we can detect both hardware and software related anomalies and events. The underlying principle for hardware faults detected by software is that only those faults are important which directly impact software performance, hence they have a marked impact on software metrics.

One important factor which has to be taken into account is that these types of fault show "symptoms" in the software usually much later that they actually occur. Usually, a trade-off is made between those anomalies which are easier to detect versus the ones which are more difficult to detect. This allows for a more reliable root cause analysis which leads to a more robust and reliable self-healing and reconfiguration methods.

Prediction of when a anomaly might occur is extremely useful for scheduling and reconfiguration. If we can predict in a timely manner that an anomaly will most likely occur on a node we can prevent task failure by not scheduling it execution on the failing node. Many types of anomalies have been shown to be detectable via monitoring software such as; multiple corrected memory errors, prediction of HDD failures (based on SMART metrics).

Metrics from software representing events can be clustered into distinctive types which can be used for stochastic propagation chains [3,5,6]. Optimally these would allow us to detect precursors anomalies before an actual fault event is about to take place. However, this is not the case for some types of fault which do not have

precursor anomalies. Currently in the available research there is not clear answer for improving predictive model recall.

Pure software anomalies are more common than hardware related ones. They represent as much as 70% to 90% of the total anomalous event from a system. In addition to this these types of anomalies are harder to categorize then hardware anomalies. The majority of software anomalies occur during the development phases of an application. So quite a large proportion of these can be classified as unhandeld exception, coding errors, logical errors etc. Exascale systems will be even more predisposed to these types of anomalies. In particular concurrency (bottlenecks, interaction failures between tasks etc.) and performance (insufficient computational resources, timeouts) anomalies.

It is easy to see that most of the pure software related anomalies are characterized by a specific behaviour which yields contextual and sequential anomalies. The research being done for these types of anomalies is in many ways in its infancy. Most of the time these anomalies are not tested in production systems as it is difficult to characterize the behaviour of a system in the long run. Production systems have varying workloads, changing usage patterns (i.e. behaviour) which at first glance can be anomalous however, it is still could be a valid scenario.

Scenarios

In the previous sections we have detailed data locality and how it can be effected by events and anomalies. In particular we focused on hardware and software faults. The former type of anomalies are easier to categories. We can easily check for metrics and features for analysis. Moreover, in some scenarios hardware faults have certain precursor anomalies which we can detect and act on this knowledge regarding data locality and scheduling of tasks.

Scenarios regarding software anomalous events and faults is much more difficult to characterize. It is extremely difficult to define generic enough anomalies which could be used for different use cases and workloads. Software and application anomalies tend to be more complex and require greater precision from any predictive model.

Data locality will be most affected by hardware failures. These, once detected, will necessitate several actions to be performed. These actions include rescheduling of tasks, moving data from a node which exhibited anomalous behaviour or was predicted to have a high probability of failure, adjust data replication etc. In the case of software anomalies the actions are not so clear cut when it comes to data locality. Additional research is required in this direction dealing in particular with root cause analysis of software anomalies for Exascale. This type of research will require high quality historical monitoring data containing both software and hardware labeled anomalies from a massively distributed system running parallel workloads. Such a

dataset will accelerate the research in this field similarly how the ImageNet[3] dataset did to the field of computer vision.

2.2 Bull IO Instrumentation

Bull IO Instrumentation is an intuitive job level I/O profiling tool. It is a scalable tool that helps the system administrators, developers and product support experts analyzing the IO activity generated by HPC jobs. A set of IO related metrics, such as IO volumes, IO response times, etc., are collected, stored and displayed in a GUI Web application.

Bull IO Instrumentation consists of two main parts:

- The IO metrics gathering chain, composed of:
  - an iolib library intercepting IO related calls to the glibc, and a job agent running on compute nodes.
  - a set of gateway services, receiving and aggregating IO metrics from the compute nodes, and storing them in a distributed MongoDB database. A gateway can serve a compute island up to 300 nodes and is typically instantiated on a service node.
  - a master service, in charge of storing job descriptions data such as job ID, start time, etc.
- A GUI (Graphical User Interface) and an API (Application Programming Interface), composed of a Web server and an application accessing the MongoDB database and creating comprehensive views.

Bull IO Instrumentation is integrated into SLURM through plugins that automate the instrumentation process. Once Bull IO Instrumentation is installed and set up, any job submitted to SLURM using the command line option '–ioinstrumentation=yes' is automatically instrumented. Note, the plugin can be configured to set the default option to yes. In this case the monitoring is activated by default for all the running jobs.

Once the monitoring tool is activated the IO activity generated by HPC jobs can be analyzed. A set of IO related metrics are collected and stored. We can identify 4 mains basics categories of metrics:

- the IO sizes
- the IO durations
- the IO patterns

[3] http://www.image-net.org/

- the number of accessed files

Each one of these categories of metrics is detailed for the read and write modes. From these 4 basic categories, some additional metrics are built (still for the read and write modes), like:

- the number of IO
- the total volume of IO
- the data locality (cache miss or hit)

Each metric is collected at a compute node level for each 5 seconds timeframe. So the data activity along time can be analyzed by compute node, by compute island or for the whole set of nodes used by the job.

Additional information on jobs such as job name, environment, node list, start time, etc. is collected and stored in the database as well.

The IO Instrumentation product is also able to collect asynchronously event-based data. Three type of events have been identified to be collected:

- user triggered
  - to record steps in workflows (CLI to call on compute nodes)
  - to record interesting timestamps when analyzing jobs in GUI
- application environment related
  - Lustre client being evicted, Lustre cache full (Max Dirty), ...
  - memory full, swapping, ...
- internal to IO Instrumentation
  - metrics lost, communication error, ...
  - start/end of processes, ...

Note this feature of event collection is currently under development and will be available in the next release of the Bull IO Instrumentation product expected for mid-2020.

2.2.1 Post-mortem vs In-vivo IO monitoring

All the IO metrics and events collected by the IO Instrumentation allows to analyse the IO activity of a job. Although the mainstream usage of the IO Instrumentation is to investigate on the IO behavior of a job in a post-mortem analysis, the collected data can also be used for in-vivo monitoring.

Indeed, the collected data are stored on the fly in the database, being available while the job is still running. The IO Instrumentation can thus be used as an in-vivo monitoring system with a 5 second sample time for IO metrics (event data are asynchronously collected).

Because of the internal monitoring mechanism implemented, the collected data are actually available in the IO Instrumentation database with a delay (approximately 30 seconds). This does not permits to envision real-time in-vivo monitoring but what we can call a delayed in-vivo monitoring.

2.2.2 Infrastructure IO monitoring

The IO Instrumentation approach is application-based, i.e. the data are collected while a job is running and they are associated to this job. Nevertheless, if, for all HPC jobs running in the machine, the IO Instrumentation is activated, we can consider the whole IO activity is monitored. Indeed, the major part of IO activity is generated by the running jobs, while we expect to have an IO activity close to zero when no job is running.

In this case, the collected data of all jobs can be summed-up at each time-frame to obtain an infrastructure view of the IO activity. Indeed, the 5 seconds timeframe sample period is not aligned with the job start time, but with the supercomputer system clock. The first and last time slice records of a job usually contain less than 5 seconds of job activity, but it allows to sum up time slice records from different jobs, to display a consolidated view of the overall supercomputer IO activity.

While this coarse overview of the IO activity is built from job-level information, different scale of analysis are possible. The data locality information can be extracted crossing the data from the IO metrics, the metadata of jobs (e.g. nodes used), and the events captured.

Chapter 3

Hardware discovery techniques

This chapter presents the process of matchmaking tasks, data and computational nodes. In order to provide the functionalities described in Chapter 4 the scheduler needs to be aware not only of the state of the data, but also of the hardware features and availability of each node. A task may depend on or prefer several features such as number of cores, amount of memory, disk space, or GPU cards. Additionally, anomalies described in Chapter 2 may trigger the restart or movement of tasks between nodes during the workflows lifecycle.

In addition to assessing hardware characteristics (cores, memory, disk), SLURM allows for each node to declare a set of user defined features that the **ASPIDE** platform can use to enhance the matchmaking process. Conversely, a task is able to use the features to constrain the scheduler to choose specific architectures that enhance the task execution.

3.1 Resource reservation

Centralized scheduling techniques may be overwhelmed by the amount of data analysis needed to assign a set of tasks to a subset of compute nodes and has been investigated for warehouse scale computers [7]. Thus, several distributed schedulers have been proposed across the years: Apollo [8], Sparrow [9], Hopper [10], none of which consider data locality or task dependency. A decentralized data-locality aware mechanism using back-pressure for task routing has been studied on peer-to-peer systems for MapReduce applications. [11, 12].

There are to two options for distributing scheduling operations:

- pessimistic: each scheduler has access to a partition only, in this case no conflicts can appear

- optimistic: each scheduler can schedule on any resource, and conflicts require the rescheduling of some tasks

In **ASPIDE**, several schedulers will be used in order to load balance scheduling decisions. They will share the view of the whole system making use of the **ASPIDE** monitoring tools.

In the case of **ASPIDE** applications, computing nodes must be selected such as to minimize data movement for the entire workflow of the application. For example, in the case of pipelined parallel tasks, it may prove more efficient to transfer data on some nodes for the first step, but then take advantage of the local reads and rack interconnectivity for the subsequent synchronization steps and file transfers. Nevertheless, hardware failures and anomalies may lead to some tasks being moved and thus degrading the expected optimization. In the case of such an event, the scheduler needs to evaluate the assignment options again.

Given a set of nodes, N, and a DAG of tasks, T, Algorithm 1 will find an assignment A, such that for each task, data movement is minimized.

Algorithm 1: Resource reservation algorithm.

```
Input: N – set of nodes, T – DAG of tasks
Result: A – mapping of task on compute node
A = {};
foreach task in T do
    nf = filter(N, task.features);
    nd = filter(N, task.data);
    min_score = MAX_FLOAT;
    foreach node in nf do
        score = min_distance(nd, n, task.data) ;
        foreach dep in task.dependencies do
            if A[dep] != n then
                score+= distance(A[dep], n, task.data),
            end
        end
        if score < min_score then
            min_score = score;
            A[task] = n
        end
    end
end
```

For each task, two subsets of compute nodes (Carea) are filtered: the available ones that meet the feature requirements of the tasks, nf, and nodes that hold the data required by the task, nd. Then, each node, $n \in nf$, receives a score based on how close it is to the data and the task dependencies. This score accounts for the

minimum $distance$ between n and any node hosting the required data and the sum of distances between the node and all task dependencies of the current task. If the node holds the data or is host to a dependency, then the distance is 0. Otherwise, is computed based on Equation 3.1

$$dist(n, m, data) = \frac{data.size}{bandwith(n, m)} \tag{3.1}$$

where $bandwidth(n, m)$ measures the connectivity of two nodes and can reflect the difference between intra-rack and inter-rack data transfers.

The score function is summarized by Equation 3.2

$$score(n, t) = min\_dist(n, nd, t.data) + \sum_{i=1}^{n_{dep}} dist(n, A[t_i], t_i.out) \tag{3.2}$$

If an anomaly will trigger a task to be restarted, Algorithm 1 will be employed again, starting from the task that has failed. In case of a hardware failure and data was lost during an intermediary step, the task that generated the data (the dependency) will need to be restarted as well, deferring scheduling up the application DAG.

Additionally, the score can also take into account the distance between the tasks of the same DAG, even if there are no dependencies between them, with the purpose of keeping the entire workflow more tightly connected.

Since assignments are known for all tasks during the DAG execution, the output of one task can staged to a node assigned for the next step during the generation of the data. The workload isolation described in Chapter 2.3 of Deliverable D4.2 [13] allows for writing or replicating output data to a specific node. Thus, if the system state did not allow of data locality during resource assignment, this solution is improved by having the data ready at the remote node shortly after the execution of workflow stage.

3.1.1 Resource sharing

In order to improve resource utilization resource sharing can be employed at task level. A task can run alongside others on the same node, if their other requirements do not exceed the node capacity. In this case, the filtering function can be relaxed to include busy nodes with enough available resources (cores, memory, disk) to run the task. In this case, the score function can be adapted to achieve different goals:

- task compaction – run as many tasks on the minimum amount of nodes
- task fairness – prefer idle small nodes over busy large nodes

- progressive – as the resources become scarce, compaction is preferred

In order to achieve task compaction, the score can be computed as the difference between the task requirements and node available capacity. In this case busy nodes are getting work before others, and thus their resource utilization increases. A task fairness policy will imply resource fragmentation, but will ensure that a task is not affected by the execution of another task on the same node. A mix of the two can be used to employ a progressive policy, where task fairness is preferred when the system is not loaded, and use task compaction proportionally to the load of the system.

Chapter 4

Exploiting metrics in the data-centric programming model

In this chapter, we present the high-level interfaces for creating tasks and task pools to be scheduled and executed by the runtime. In **ASPIDE**, we focus on improving throughput and performance by increasing the data locality when assigning the task to a given processor. To do so, the scheduler may require detailed information about the platform and metrics like the load status of the nodes to better decide where to run the different tasks. In this sense, multiple scheduling policies can be eventually implemented to choose different parameters for placing the tasks. For instance, if the nodes are not overloaded a good solution is to place the tasks where the data is, but, if a node is overloaded it might be better to move the data to an underused node to increase the performance. However, to perform this node selections, some information will be required by the scheduler. In previous chapters, we have defined the necessary metrics and the hardware information that will be collected by the monitoring and hardware discovery modules. In this chapter, we also present the interfaces for accessing and obtaining that information for the runtime scheduler.

4.1 Tasks and data interfaces

In this section, we describe the interfaces designed for creating tasks and pool of tasks at a high abstraction level in GrPPI and for gathering information about the platform to inform the scheduler. With this information, the scheduler may take a better decision for placing the tasks by improving data location and performance.

4.1.1 Task

To support the DCEx model, we define the task interface as the minimal concurrent activity that applies a transformation code to an input data set to produce another data set. To better explain the interface, first, we define the structure that represents a task. This structure will univocally identify a task and will store the necessary information to be scheduled. The task structure is comprised of the following attributes:

- name/ID: identifies a task and it is introduced by the runtime.
- data_layout: specifies the data location and configuration information of a dataset.
  - data_location: identifies the node that stores the corresponding dataset which is provided by the intelligent data manager.
  - policy: specifies the data placement policy and if it is replicated on different nodes.
  - data_source: determines the URI of the dataset
- function_code: stores the code that should be run on the input dataset to produce the outputs. This function code can be a file that contains the source code or a pointer to a function.
- argument_list: stores the list of additional arguments required by the task.
- number_threads: determines the number of threads that will be used by the task. If the task code is sequential, this number will be 1.
- dependencies: stores the dependencies with other tasks.

Listing 4.1 shows the constructor interface of a new task in GrPPI. This constructor receives as the two firsts arguments the input and output data sources and if needed those arguments can also specify the configuration information of each data set by using the container interface. The third argument is the function code that can be a C++ callable object/function pointer or the name of a python file. Next, this constructor receives a tuple that contains the list of additional arguments, the number of threads and the dependencies that are stored as the list of the task IDs.

Listing 4.1: "Task interface."

```
Task t = grppi::task(
input_data,
output_data,
function_code,
argument_list,
num_threads,
```

```
dependencies
);
```

4.1.2 Pool

A task pool defines a set of tasks that can run in parallel if all the dependencies are satisfied, otherwise, they will wait until the required data becomes available. This can be done by creating a list of tasks and setting up the dependencies among them to build the DAG that determines the task parallel execution. However, this is quite complex for not experienced programmers. To ease the creation of multiple tasks with different dependencies, we take advantage of well-known parallel patterns. These patterns present different dependency patterns that match with a wide range of scientific and industrial applications. In the following, we describe the interface for some of the parallel patterns that will be supported in this project.

The first pattern is the pipeline, which represents a sequence of tasks that are applied consecutively on each dataset or data item part of a data collection/dataset. In this pattern, each task establishes a dependency on the previous task that will produce its input. Therefore, tasks executing different stages over different data can be executed in parallel, but only a single worker entity can run a given stage at a time. This is given by the fact that each stage may use some state information that can produce side-effects.

Listing 4.2 shows the interface of a pipeline pattern. This pattern receives as a first and last argument a list of data sources (DataCollection) that identifies each data set to be processed and the URIs to store the produced data, respectively. The rest of the arguments are the task that will be executed over each data set. The interface of this pattern also hides some complexities related to the task creation by not requiring the definition of the input data nor the task dependencies. Basically, for the first task, the input data will be one of the data set parts of the input data collection, while for the rest of the task in the pattern the input will be the output of the previous task.

Listing 4.2: "Pipeline interface."

```
grppi::pipeline(DataCollection in, Task T1, ...,
Task Tn, DataCollection out);
```

The farm pattern represents a set of task that does not have any dependency among them, and, therefore they can be executed in parallel over different data sets/data items. Listing 4.3 shows the interface of a farm pattern. This interface receives the input data collection/dataset, the task that represents the code to be applied over each data set and the output data collection. Basically, this interface will create a task for each data set/item and they will be run in parallel, storing the results in the

corresponding data set for the output data collection.

Listing 4.3: "Farm interface."

```
grppi::farm(DataCollection in, Task T,
DataCollection out);
```

Similarly, the accumulator pattern represents a pool of tasks that reduces subsets of data items in a dataset or data collection to produce new data items. This pattern uses a given reduction function to generate a data item that collapses several input data items. The accumulator interface, as shown in Listing 4.4 receives, apart from the input and output data collections/datasets, the window size, i.e., the number of items that will be part of each reduction operation; the offset, determining the number of overlapping items among windows; the initial value of the reductions; and the reduction task (`combine_op`). In this case, the concurrent entities in the parallel implementation are responsible for processing individually the accumulation of each input window.

Listing 4.4: Accumulator interface.

```
grppi::accumulate(DataCollection in, int window_size,
int offset, InitVal initial_value,
Task combine_op, DataCollection out);
```

Another pattern that will be supported in this project is the filter pattern. This pattern selects the data items that will be part of the output dataset/data collection depending on a predicate function. In this case, as described in Listing 4.5, we provide two interface alternatives *keep* and *discard* depending on which items are filtered out. If the *keep* function is used only those items that the filter evaluates as true are introduced on the output data collection/dataset. On the contrary, *discard* removes those items evaluated as true.

Listing 4.5: Filter interface.

```
grppi::keep(DataCollection in, Task  predicate_op,
DataCollection out);

grppi::discard(DataCollection in, Task predicate_op,
DataCollection out)
```

Finally, the well-known map-reduce pattern represents a transformation function followed by a reduction function that produces a data set with a reduced cardinality. Basically, the input data is divided into different subsets to which the transformation function is applied to produce a set of key,value. Then depending on the implementation, the resulting data subsets may be shuffled and then all the resulting key,value

data sets are reduced to produce a single one. Listing 4.7 shows the interface of the map-reduce pattern. This function receives, as the first argument, the DataCollection that will be divided into several data sets. Then it receives the transformation function that produces a data set in the form of key,value data elements and the reduction function that given two data sets produces a new data set. Finally, the last argument is the output DataCollection that will store the resulting dataset.

Listing 4.6: "MapReduce interface."

```
grppi::map-reduce(DataCollection in, Task Map,
Task Reduce, DataCollection out);
```

In any case, all of those patterns can be composed within a pipeline pattern. If so, the input and output data collections/datasets are omitted in the inner patterns, since they are determined by the stage outputs of the previous stages. This way, parallel pattern compositions allows users to define multiple workflow patterns. Listing **??** shows an example of a pattern composition using the farm patters as a given stage. In this example, the second stage generates task as farm pattern, therefore, multiple instances of tasks coming from the map pattern can be executed in parallel, i.e. different worker entities can run the second stage on different input data in parallel, while the first task can be only executed by a single worker at a time.

Listing 4.7: "Pipeline composition example."

```
grppi::pipeline(in_datacollection,
task_stage1,
grppi::farm(task_stage2),
out_datacollection);
```

In this sense, to build more complex constructs, we define the split-join operator. This operator can be only used within a pipeline pattern and it is applied in different phases. In the first phase, the input data is divided into different substreams, following a given spiting policy, which can be processed in parallel. These substreams may apply different transformations to the data that can be defined by a single task o a new parallel pattern. Afterward, the results of each substream are joined into the mainstream by using a given combining policy. In this project, we plan to support the following splitting and combining policies:

- Splitting policies:
  - **Duplicate:** this policy passes the same input data item/dataset to all of the substreams defined on the split-join operator.
  - **Round-Robin:** in this case a data item is passed to one of the substreams following a Round-Robin policy.
- Combining policies:

- **Round-Robin:** the items delivered at the end of the substreams are combined into the main data stream employing the Round-Robbin policy. In other words, each output of the split-join operator corresponds to one of the substream outputs.
- **Tuppler:** this policy combines the results of the substream by creating a tuple that contains the output of each substream.

The interface designed for the split-join, shown in Listing 4.8, receives the split and join policies (`split_policy` and `join_policy`) and the list of tasks/pattern (`transformers`) that should be applied to the different output streams.

Listing 4.8: Split-join interface.

```
grppi::split_join(SP split_policy, JP join_policy,
T ... transformers);
```

4.2 Metrics collection

In this section, we describe the interfaces for collecting the information about the available hardware in the platform and the status of the system at a given moment. Additionally, we briefly describe the interface for obtaining metadata regarding the data required for a given task in order to exploit data locality.

Listing 4.9: "Hardware information interfaces."

```
hw_info = getHWInfo();
node_stat = getNodeStats(node_id);
sys_stat = getSysStats();
```

Listing 4.9 describe the functions related to the monitoring and hardware discovery modules. The first function ("getHWInfo") obtains the information about the available hardware on the different nodes in the platform. This way the information about total available memory or the existence of a given device such a GPUs may affect the assignment of a task to a given processing unit. For instance, if a task requires a GPU, thanks to this information, the scheduler will only choose a node that has a GPU.

The second function and third functions, related to the monitoring module, obtain the metrics for a given node (getNodeStats(node_id)) and for the whole system (getSysStats()) to be used by the scheduler. Using this information, the scheduler may choose to move a task from a node that does not have enough free memory o CPU resources to another node that is being underused. This way, we can

promote performance by moving a task to nodes whose resources are not being fully exploited.

On the other hand, in order to improve the data locality on the scheduler, we need to define two functions to let the scheduler know where the data is and the size of that data. Listing 4.10 describe the functions to obtain this information. First, getDataLocation function obtains the id of the node or list of nodes (if the data is replicated) that stores the information identified by "data_id". With this information, the scheduler will send the task to a node that already has the data to minimize data transfers among nodes and, thus, improving the data locality and performance. The second function obtains the size in bytes of the dataset identified by "data_id". This information is intended to be used along with the hardware information to not launch a task that does not have enough memory resources to allocate the data, i.e. if the data size is greater than the node available memory.

Listing 4.10: "Data information interfaces."

```
loc = getDataLocation(data_id);
size_t = getDataSize(data_id);
```

Chapter 5

Data placement mechanisms

This chapter describes how our proposed AIDE-based solution has been integrated with the proposed programming model for efficiently supporting many-task workflow applications and how it maps with the special requirements of data-intensive applications.

For achieving *server scalability*, we deploy storage back-ends on-demand on every node (one instance per node, not per worker) before the application is started. The main advantage of this approach is that the total memory scales with the number of worker nodes and it is available to applications as a cooperative distributed cache. In contrast, the shared file system suffers from two main problems. First, the client caches are non-cooperative. When the locality is not exploited, the exchange of data between tasks happens through file system server caches. Second, there is a limited amount of cache at a fixed number of file servers. Thus the access performance is likely to degrade with a substantial increase in the number of processes/tasks as a result of contention caused by both metadata accesses (large number of files) and data accesses (larger data volume). Nevertheless, this omnipresence of servers on every application node allows each client to potentially benefit from having data stored locally on the same node, as discussed below.

For enabling *locality exploitation* by workflow engines running on top of AIDE, we leverage the omnipresence of servers on all application nodes and the separation of data and metadata management discussed above. For exposing and exploiting data locality, we offer two new API calls:

- int `get_rank(filename)` a function that returns the rank corresponding to a server where a given file is cached if the file exists
- void `set_rank(filename, rank)` a function that enforces a future file to be created at a server running on a *Cnode* with given rank

These functions map directly with the lookup and forced placement file creation

functions introduced in Section **??**. The `get_rank` function allows a workflow engine to locate a file. This information can be used for scheduling a task close to the data. The `set_rank` function allows placing the data on a given server based on user-specified criteria. This function can be used to control the load or capacity balance of a back-end servers or worker nodes. These functions rely on looking up the file metadata at a server identified by hashing the file name and getting or setting the metadata accordingly.

Since our solution targets intermediary files whose life cycle consists of being entirely written, followed by being read in their entirety, followed by deletion, there is no need for costly locking algorithms to enforce consistency, as in the case of general-purpose parallel file systems such as GPFS [14] and Lustre [15]. Sequential consistency is naturally enforced by internally reading a buffer after write completion.

5.1 Task placement for supporting locality-aware dynamic scheduling

The data placement functions proposed above are key components for enhancing data-locality. However, data placement has to be complemented with task placement management in order to open up an optimization space that can be used for trading off load balance and locality.

In Deliverable 2.1, we presented the main components of the DCEx programming model, defining the two basic constructs to refer to computing nodes and computing areas:

- *Cnode* representing a single computing node, and
- *Carea* representing a region (or area) including a set of computing nodes.

As claimed, *Cnodes* and *Careas* are used to implement data and task locality by specifying a mapping between data loading operations and the task execution operations (introduced in Chapter 4).

The typical default scheduling scheme in workflow engines is data locality agnostic. In this mode, the task is first placed in the work queue of the *Cnode* where it is created. If no work stealing events are triggered, the task remains there until a worker attached to the server obtains the task. If another server steals the task, it can be executed on any worker in the system. Thus, this scheme is not predictable, but it allows total flexibility to the load balancer. However, the application of this scheme in a data-intensive use case would result in a great deal of network traffic since locality is not applied.

In order to allow applications to take advantage of data locality, we have worked

```
 1: task (file file_output) write_file(){
 2:  "./write-local.sh" file_output;
 3: }
 4: task (void) read_file(file file_input){
 5:   "java read-local.class" file_input;
 6: }
 7: foreach i in [0:n-1] {
 8:   set_rank(filename[i], i % nnodes)
 9:   rank1 = get_rank(filename[i])
10:   location L1 = location(rank1, SOFT, RANK)
11:    file f<file_name[i]> =
       @location = L1 write_file();
12:   rank2 = get_rank(filename[i])
13:   location L2 = location(rank2, SOFT, RANK)
14:   @location = L2 read_file(f);
15: }
```

Figure 5.1: Example of a data-parallel workflow, in which files are distributed in a round-robin way over available ranks (line 8). A best-effort task placement policy (SOFT) is used for placing both the write_file and read_file tasks on the same node with the file (lines 11 and 14).

in the implementation of new task placement functionality integrated in the AIDE framework. AIDE offers the ability of placing tasks on given ranks (specific workers) or *Cnodes* (*n* workers can be running on the same node), which combines with data-placement mechanisms for exposing data-locality enables the possibility of exploiting task and data co-location.

This is achieved by using a special type of variable called `location` and task annotation, as illustrated in Figure 5.1. Line 8 enforces that file i is created on cnode $i\%nnodes$, where $nnodes$ is the number of *Cnodes* on which the program runs and line 9 returns a rank running on the *Cnode* where the file is stored. Line 10 initializes a `location` variable, which is used in line 11 in a best-effort policy for placing the task locally to the data. Lines 12-14 have the same functionality for `read_file` task as lines 9-11 for `write_file` task. By commenting out line 8, the program uses the default data placement. By commenting out lines 8-10 and the location construct from line 11, the program uses the default data placement and the default task placement for `write_file` task.

The `location` type has three attributes. The first one is the *Cnode* id on which a task will execute, identifying a specific worker. The second argument indicates whether the task placement is strict (`HARD`, used in the example) or advisory (`SOFT`). The third argument indicates whether the placement has to be done on the exact rank given by the first argument (`RANK`, used in the example) or on the *Cnode* on which the worker process `rank` is running (`NODE`). The value `NODE` gives to the

AIDE load balancer more flexibility to place the tasks. The Cartesian product of the last two variables provides four modes that can be used for task placement:

- HARD RANK enforces the task placement on the given *Cnode* id (worker).
- HARD NODE enforces the task placement on the *Cnode* where the process with the given rank runs and dynamically selects the best-available worker in the *Cnode*.
- SOFT RANK offers best-effort task placement on the given id (worker). The load balancer can decide to move the task on another worker running on the same *Cnode* or on a different *Cnode*.
- SOFT NODE offers best-effort task placement on the *Cnode* of the given rank. The load balancer can decide to move the task on a worker running in a different *Cnode*.

HARD RANK scheme offers predictable behaviour. Load balancing, however, suffers greatly as tasks are forced to execute on a specific worker. This feature could be used selectively to perform application operations that strictly require access to a node-local file or some specific component (e.g., a node-local database). It could be useful in a system without Hercules or another data movement technique. However, it falls far short of our complete system with Hercules.

The other three schemes successively relax the task placement, letting the load balancer more freedom to improve the overall system load balance. The most relaxed scheme is SOFT NODE. In this mode, tasks are given a rank on which to execute. If this rank is busy, another rank on the same *Cnode* may take the task. If these ranks are all busy, another worker under the same server may take the task. If a server is the target of a steal and all of its tasks are soft targeted, a soft-targeted task may be taken.

These last three schemes offer various degrees of a mix of predictable and dynamic behaviour. They prevent a block in dataflow progress by allowing idle workers to continue making progress even if network data movement is required, but they boost the likelihood of local data access if the necessary data locality control mechanisms described in the next section are used.

Chapter 6

Conclusions

Current scheduling schemes are data locality agnostic. In this mode, tasks are first placed in the work queue of the node where they are created. If no work stealing events are triggered, each task remains there until a worker attached to the server obtains the task. If another server steals the task, it can be executed on any worker in the system. Thus, this scheme is not predictable, but it allows total flexibility to the load balancer. However, the application of this scheme in a data-intensive use case would result in a great increase of network traffic since locality is not applied. The data placement mechanisms proposed in **ASPIDE** and integrated in the IDM component are key elements for enhancing data-locality. Data placement has to be complemented with task placement management in order to open up an optimization space that can be used for trading off load balance and data locality.

Bibliography

[1] Ariel Oleksiak; David E. Singh; Dragi Kimovski; Gabriel Iuhasz; Javier Garcia-Blas; Jesus Carretero; Vladislav Kashansky. D3.2 extreme scale monitoring architecture. Technical report, H2020 ASPIDE, 2019.

[2] Prateek Agrawal Dragi Kimovski, Gabriel Iuhasz. Monitoring data collection and mining for exascale systems - d3.3. Technical report, ASPIDE, 2019.

[3] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A Debardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, May 2014.

[4] Thomas Herault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[5] Alessio Netti, Zeynep Kiziltan, Ozalp Babaoglu, Alina Sîrbu, Andrea Bartolini, and Andrea Borghesi. A machine learning approach to online fault classification in hpc systems. *Future Generation Computer Systems*, 2019.

[6] Qiang Guan and Song Fu. Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 205–214, Washington, DC, USA, 2013. IEEE Computer Society.

[7] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, volume 14, pages 285–300, 2014.

[9] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[10] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 379–392. ACM, 2015.

[11] Weina Wang, Matthew Barnard, and Lei Ying. Decentralized scheduling with data locality for data-parallel computation on peer-to-peer networks. In *Communication, Control, and Computing (Allerton), 2015 53rd Annual Allerton Conference on*, pages 337–344. IEEE, 2015.

[12] Weina Wang and Lei Ying. Resource allocation for data-parallel computing in networks with data locality. In *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*, pages 933–939. IEEE, 2016.

[13] David E. Singh; David del Rio Astorga; Javier Fernandez Muñoz; Javier Garcia-Blas; Jesus Carretero; Silviu Panica. D4.2 early tool prototypes for data management. Technical report, H2020 ASPIDE, 2019.

[14] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, 2002. USENIX Association.

[15] Cluster File Systems Inc. Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002.