Accelerated Execution via Eager Release of Dependencies in Task-based Workflows

The International Journal of High Performance Computing Applications
XX(X):1–17

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Hatem Elshazly[1] and Francesc Lordan[1] and Jorge Ejarque[1] and Rosa M. Badia[1]

Abstract

Task-based programming models offer a flexible way to express the unstructured parallelism patterns of nowadays complex applications. This expressive capability is required to achieve maximum possible performance for applications that are executed in distributed execution platforms.

In current task-based workflows, tasks are launched for execution when their data dependencies are satisfied. However, even though the data dependencies of a certain task might have been already produced, the execution of this task will be delayed until its predecessor tasks completely finish their execution. As a consequence of this approach of releasing dependencies, the amount of parallelism inherent in applications is limited and performance improvement opportunities are wasted.

To mitigate this limitation, we propose an eager approach for releasing data dependencies. Following this approach, the execution of tasks will not be delayed until their predecessor tasks completely finish their execution, instead, tasks will be launched for execution as soon as their data requirements are available. Hence, more parallelism is exposed and applications can achieve higher levels of performance by overlapping the execution of tasks.

Towards achieving this goal, in this paper we propose applying two changes to task-based workflow systems. First, modifying the dependency relationships of tasks to be specified not only in terms of predecessor and successor tasks but also in terms of the data that caused these dependencies. Second, triggering the release of dependencies as soon as a predecessor task generates the output data instead of having to wait until the end of the predecessor execution to release all of its dependencies.

We realize this proposal using PyCOMPSs: a task-based programming model for parallelizing Python applications. Our experiments show that using an eager approach for releasing dependencies achieves more than 50% performance improvement in the total execution time as compared to the default approach of releasing dependencies.

Keywords

Task-based Workflows, Partial Dependencies, Lazy Dependency Release, Eager Dependency Release, High Performance Computing, Parallel Programming, Distributed Execution

1 Introduction

The rapid increase in computational power goes side by side with an increasing complexity in application domains. In fields of science and engineering (e.g. computational biology, molecular dynamics, mechanical turbines simulation, etc.), it is necessary to solve complex problems that exhibit irregular patterns. These patterns are characterized by their complex computation flows, access patterns and execution branches. Such problems are usually represented by complex data structures such as trees and graphs.

This increasing complexity in problem and solution domains calls for parallel programming models that are able to exploit the unstructured patterns of applications and, at the same time, hide the complexity of the underlying execution platform.

Task-based programming models allow for a flexible approach to express irregular parallelism as opposed to programming models that follow a specific parallel paradigm such as Map-Reduce Dean and Ghemawat (2008) and its alternative Spark Zaharia et al. (2016). Other parallel programming models such as MPI Gropp et al. (1999) and OpenMP Dagum and Menon (1998) are widely used. However, gaining performance using these models requires certain programming expertise. In addition to that, it exposes the details of the underlying execution infrastructure which could compromise the programmability of applications.

Using a task-based programming model, applications are decomposed into tasks. These tasks are organized in the form of a Directed Acyclic Graph (DAG) by detecting data dependencies between them so that each task has predecessor(s) and successor(s). Data dependencies between tasks control the scheduling of tasks and their execution. Tasks are launched for execution if they are dependency-free, i.e. all their predecessors have finished their execution successfully.

[1]Barcelona Supercomputing Center, Barcelona, Spain

Corresponding author:
Hatem Elshazly, Barcelona Supercomputing Center (BSC), C/ Jordi Girona, 31, 08034 Barcelona, Spain.
Email: hatem.elshazly@bsc.es

A data dependency relationship can occur between two tasks such that a task depends only on some of the outputs of its predecessor not the whole output set. Throughout this paper, we call this type of dependency *Partial Dependency*. In this scenario, regardless of whether the outputs that constitute the dependency relationship with a task are ready, a task requiring these outputs will not be executed until the predecessor task completely finishes its execution. We call this type of releasing dependencies a *Lazy Release* of dependencies.

This pattern of partial dependency relationships is common in applications that target the simulation of physical or geometrical systems or data parallel applications like text analysis or bioinformatics applications. In these applications, tasks generate multiple output data where each has independent execution pipelines.

In such application domains, a task that periodically produces output data has the potential of creating more parallelism by allowing overlapping the execution of tasks. For instance, if a task produced some data that are required by one of its successors, this successor task can be released for execution while the predecessor task is still computing the remaining outputs. Similarly, these parallelism opportunities can also occur if a task generates output data in a parallel manner, but the workload of this task is imbalanced or the executing parallel processes or threads experience performance variability, which is a common problem in parallel I/O and distributed systems domains Tan et al. (2013), Lin-Wen Lee et al. (2000).

In this context, using a task-based programming model with a lazy approach of releasing dependencies has a drawback that limits the maximum amount of achievable parallelism in applications. Successor tasks' execution will be blocked until their predecessor tasks completely finish execution even if the data required by the successor tasks have been already produced.

Our work targets the problem of the limited achieved parallelism because of the lazy approach of releasing data dependencies. In this paper, we propose an eager approach for releasing data dependencies. Using this approach, a task starts execution as soon as its data dependencies are ready, even if the predecessor task that produced these data has not yet finished execution. Adopting this approach accelerates the rate in which tasks are launched for execution, thus, more parallelism is exploited and higher performance can be achieved.

In this paper, we use the PyCOMPSs Tejedor et al. (2015) task-based programming model to demonstrate our contribution and its impact on the performance of applications. However, it should be noted that our proposal can be adopted by any system that uses data dependencies to manage executions.

The contribution of this paper can be summarized in the following points:

1. Introducing an eager approach for releasing data dependencies in task-based systems. This approach is achieved by the following proposals:
   - A proposal for identifying tasks' dependency relationships by their data requirements not by the execution status of their predecessor tasks.
   - A proposal for notifying the runtime system that a task has produced output data before the executing process reaches the return statement. Hence, dependencies can be released and successor tasks can start their execution as soon as their data requirements are ready even if the predecessor task has not finished execution.
2. An implementation of the eager approach for releasing dependencies in the PyCOMPSs framework and the evaluation of its performance with different use cases.

To the best of our knowledge, the aforementioned proposals are not supported by any of the current task-based programming models as will be presented in the related work section.

This document is organized as follows: The problem statement is described in Section 2 followed by Section 3 that formally describes our proposals for achieving the eager-release of data dependencies. Section 4 presents the implementation details of the eager approach of releasing dependencies in the PyCOMPSs programming model and runtime. Section 5 starts by evaluating the overhead of PyCOMPSs eager-release mechanisms then it presents the impact of using our proposal with different use cases that exhibit real patterns. Two of the use cases have a different rate of returning data and releasing dependencies which offers an evaluation of different performance aspects of our proposal. Whereas the third use case shows how our proposal enables streaming solutions for applications that have larger-than-memory inputs or exhibit high memory requirements. Related work is discussed in Section 6 before finally concluding the paper in Section 7.

2 Problem Statement

We define the problem in general directed acyclic graphs with vertices and edges that model the applications. In this model, a DAG $G = (V, E)$ has vertices $v \in V$ representing tasks and directed edges $e \in E$ representing dependencies. These dependencies are input and output data of the tasks. We denote $I(v)$ as the inputs of task v and $O(v)$ as its outputs.

Two vertices (tasks) $p, s \in V$ are said to have a dependency relationship if the task corresponding to the vertex s needs a data produced by the task corresponding to vertex p. More formally, a directed edge $e = (p, s)$ exists if $I(s) \cap O(p) \neq \emptyset$.

In the dependency $e = (p, s)$, task p is a predecessor of s because at least one of its outputs $o \in O(p)$ satisfies $o \in I(s)$. Also, task s is a successor task of p because at least one of its inputs $i \in I(s)$ satisfies $i \in O(p)$.

In task-based models, dependency relationships control when a task will be executed. Tasks are launched for execution if their data dependencies are met. Current task-based models specify dependency relationships only in terms of tasks and not the data that caused these dependencies. Consequently, successor tasks are launched for execution when all their predecessor tasks produce their entire output set(s) and completely finish execution.

It is possible that exists a dependency relationship $e = (p, s)$, where the successor task s does not require the whole

set of $O(p)$, instead, it depends only on a partial set of $O(p)$, such that $I(s) \subset O(p)$. We call this type of dependencies *Partial Dependency*. However, even though task s is partially dependent on task p, it will not be executed unless task p has produced all its outputs and finished execution.

Figure 1 highlights this point. Task p produces output set: $O(p) = \{out_1, out_2\}$ and tasks $s1$ and $s2$ have input sets: $I(s1) = \{out_1\}$ and $I(s2) = \{out_2\}$. The dependency relationships are defined as $e_1 = (p, s1)$ and $e_2 = (p, s2)$ where $I(s1) \cap O(p) = \{out_1\}$ and $I(s2) \cap O(p) = \{out_2\}$. Although s_1 and s_2 require two different outputs of p: out_1 and out_2 respectively, both successors have to wait until all outputs of p are produced and p completely finishes execution.

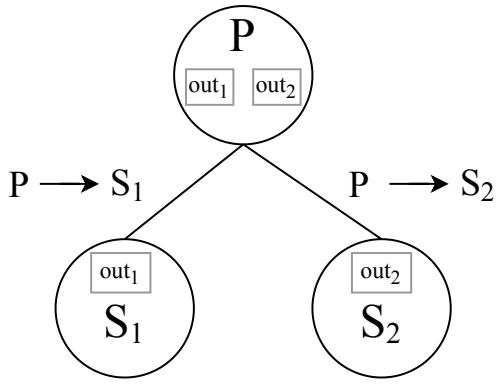


Figure 1. Dependency Relationships identified only as Task:Task Dependency

We call this default manner of releasing data dependencies: *Lazy Release*. The lazy release of data dependencies can be characterized by the inability to release a task for execution as soon as its data dependencies are satisfied. Therefore, a task is blocked and its execution is delayed until its predecessor task completely finishes execution, although the data constituting the dependency might have been ready before this point.

The lazy release of dependencies can be traced back to two reasons:

1. Data dependency relationships between tasks are identified only as task:task relationships. No information is stored about the data/parameters that resulted in the dependency relationship.
2. No mechanism exists to notify the runtime system that a dependency parameter has been generated. All the output values/dependency parameters of a task are returned when the task execution ends and the executing process of the task reaches the return statement in the user code.

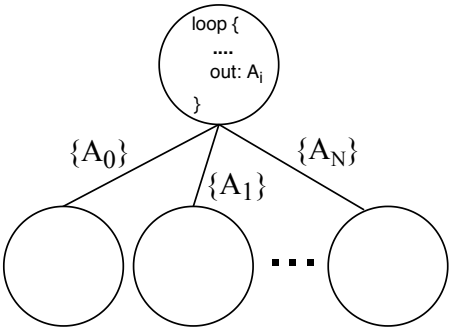
The two reasons mentioned above are intertwined. If data (dependency parameters) are generated at an early point of task execution, it will be returned after task execution ends. Even if the runtime was made aware of the availability of a dependency parameter, it will not be able to the release the tasks dependent on that parameter.

This approach of releasing dependencies will work fine if a successor task depends on the whole output set of its predecessor or if the predecessor generates the dependency values in a fast rate. However, the limitations of this approach will start to appear if a successor task has a partial dependency with its predecessor and this predecessor spends time between the generation of each output/dependency parameter. This time spent between calculating different values creates a window for more parallelism opportunities by overlapping the execution of tasks.

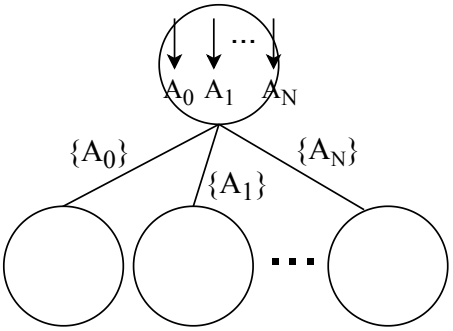
An example where the lazy approach of releasing dependencies can hinder the performance of applications: a 1:N task dependency graph where successors have a partial dependency relationship with the predecessor (the predecessor task generates N output parameters, each one feeding one of the successor tasks). In this scenario, the predecessor task (called *generator*) generates data and its N successors (called *consumers*) each partially depends on some of these data.

Figure 2 shows two possible 1:N task graphs. Figure 2(a) shows a sequential generator task which uses a loop to generate a value at each iteration. This generator task has N successors where successor *i* depends on a value generated at iteration *i*. A consumer depending on a data generated at iteration 0 will not be released for execution until the generator task completes its N iterations and finishes its execution.

Similarly, Figure 2(b) illustrates a MPI parallel task with multiple consumers where each consumer depends on the output of one of the MPI processes. Regardless of how fast one MPI process generates its data, the execution of consumer tasks is blocked until the data generator task completely finishes execution.



(a) N Consumers Partially Dependent on a Sequential Task



(b) N Consumers Partially Dependent on a Parallel MPI Task

Figure 2. Examples of 1:N Partial Data Dependency Relationships

Indeed, in some cases the example graph in Figure 2(a) can be modified to remove the 1:N dependency pattern and replace it with task parallel N:N pattern. However, such modification may introduce overhead due to N tasks creation and management. In addition to that, workflow modification may not be possible in certain scenarios where the producer task cannot be split. For instance, if the predecessor task is executing a legacy code or calling an external binary that cannot be modified. Additionally, if the predecessor task is a streaming task where it receives data from a stream then distributes these data to its successors.

3 Eager-Release of Data Dependencies

In this section, we present our proposals for eagerly releasing data dependencies in task-based models. An eager-release of data dependencies will exploit the parallelism possibilities inherent in applications. Following this approach, tasks are launched for execution as soon as their data dependencies are produced without having to wait the predecessor task to completely finish its execution. Hence, the release of tasks for execution is accelerated and more performance can be achieved.

The mechanism for eagerly releasing data dependencies can be achieved by applying two modifications to task-based systems. Each modification offers a solution to one of the shortcomings in the system design that were addressed in the previous section. These two modifications are:

- Modifying the specification of dependency relationships to include the parameters that caused the dependencies (Section 3.1).
- Triggering the release of dependencies without having to wait the predecessor task to finish its execution (Section 3.2).

Indeed, these two modifications are entwined. A task-based system that is aware of the data that caused a dependency between two tasks, will not be very useful if a successor task has to wait until its predecessor completely finishes execution. Likewise, notifying a system that a task has generated an output will not be useful if the system cannot identify which are the successors that require these data.

3.1 Parameter-Aware Dependencies

The first necessary step to achieve the eager-release of dependencies is changing the dependency specification. The system needs to be aware of which are the data/parameters that have resulted in dependency relationships between tasks. This way, successor tasks will depend on whether the required data has been produced by predecessor tasks instead of depending on whether the predecessor tasks has finished execution.

In order to achieve this goal, it is not sufficient to specify data dependency relationships in terms of the tasks themselves. Instead, tasks dependency relationships need to be identified in terms of the data that have resulted in these dependencies. We call this manner of dependency relationships specification: *Parameter-Aware Dependency*.

Formally, given a DAG $G = (V, E)$ where $i, j \in V$ are vertices representing tasks and $e \in E$ is a directed edge representing a data dependency between the predecessor task i and the successor task j. A parameter-aware specification of the dependency relationship e would be expressed as:

$$e = (i, j, d_1, d_2, .., d_n)$$

where $I(s) \cap O(p) = \{d_1, d_2, .., d_n\}$ and $n \leq |O(p)|$.

By adopting a parameter-aware approach for specifying dependencies between tasks, data are not only used by task-models to detect dependencies between tasks, but also, these data are explicitly included as part of the dependencies specification.

Figure 3 depicts the parameter-aware specification of the dependency relationships in Figure 1. In a parameter-aware model, the dependency relationship between the predecessor task p and the successor tasks s_1 and s_2 can be expressed as: $e_1 = (p, s_1, out_1)$ and $e_2 = (p, s_2, out_2)$ respectively, as opposed to their previous tasks-only specification $e_1 = (p, s1)$ and $e_2 = (p, s2)$. Now that each dependency relationship is identified by the data that caused it, the system should release tasks s_1 and s_2 as soon as their data requirement is ready (i.e. produced by task p) regardless of the execution state of task p.

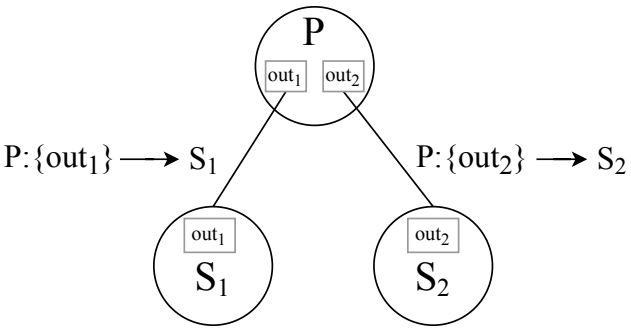


Figure 3. Parameter-Aware Dependency Relationships

3.2 Triggering The Release of Dependencies

In traditional task-based systems, the termination of task execution and releasing the data dependencies are considered as two dependent steps. First, a task has to finish its execution, identified when the executing process reaches the return statement in the task code. Then, once the task execution ends, the runtime system receives all the output values included in the return statement and starts to release the successor tasks that require them, if these successor tasks do not require more output values from other tasks.

In order to take advantage of modifying the specification of data dependency relationships to be parameter-aware, it is necessary to make task-based systems aware that output data are ready as soon as they are produced instead of waiting until the end of task execution. Otherwise, a parameter-aware dependency will have the same behaviour as a traditional tasks-only dependency specification since all the dependencies will be released after the task completely finishes the execution.

To this end, we propose extending the programming model of task-based systems to enable the notification of the runtime system whenever a task produces output data without having to wait reaching the return statement in the

task code. Thus, triggering the release of data dependencies and launching successor tasks that require these data without waiting to the end of the predecessor task execution.

The proposed extension is to include an API that can be used in the task code to allow users to notify the runtime system that output data are ready. Therefore, enabling users to plan and optimize the execution of their applications by choosing and prioritizing when the dependencies should be released during tasks execution.

Indeed, the runtime-notification API can be used at any point in the task code before the return statement. Every time such API call is encountered by the process executing the task code, the runtime system is made aware that data was produced. Once the runtime system is notified that a task has produced output data, it releases all successor tasks that require these data if the rest of their data requirements is satisfied. Meanwhile, the task execution process will resume task code execution as normal until it encounters the return statement.

Figure 4 shows the changes in the task graph state when releasing output data during task execution. Given a task graph $G = (p, s_1, s_2)$ that has a predecessor task p and two successor tasks s_1 and s_2. The parameter-aware dependency relationships of this graph can be specified as: $e_1 = (p, s_1, d_1)$ and $e_2 = (p, s_2, d_2)$ where tasks s_1 and s_2 require data d_1 and d_2 from task p respectively. Using parameter-aware dependencies and triggering the release of data dependencies, s_1 and s_2 can be released for execution as soon as their data requirement is produced instead of unnecessarily waiting until the end of task p execution. At time T_1 of task p execution, data d_1 is produced and the runtime is notified that d_1 is ready so it releases task s_1. The same behaviour is repeated at time T_2, the runtime releases task s_2 because it was notified that task p has produced data d_2. The execution of task p continues until it reaches the return statement in task p code.

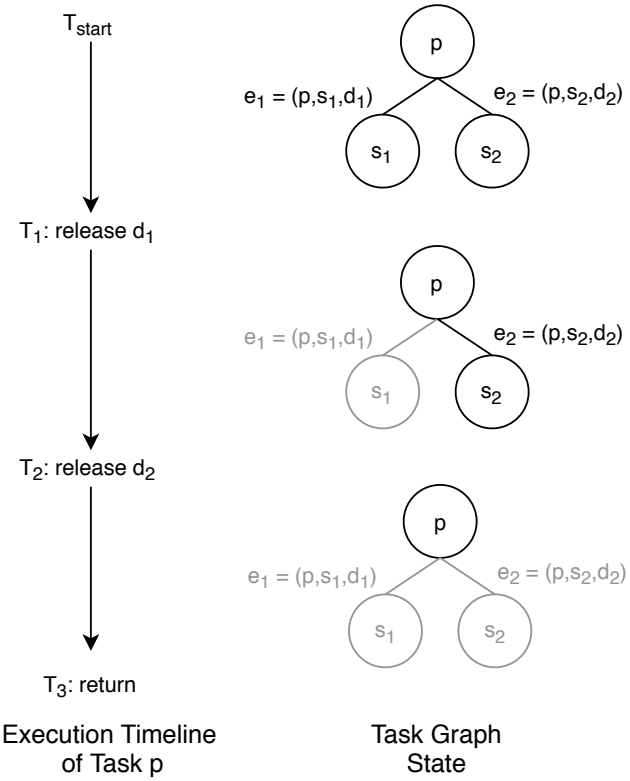


Figure 4. Releasing Data Dependencies once Data are Produced

4 Implementation

In this section, we first give a brief overview of the PyCOMPSs programming model and its runtime system in Section 4.1. Next, in Section 4.2, we present the implementation details of the eager-release of dependencies in PyCOMPSs.

4.1 PyCOMPSs Overview

PyCOMPSs is a framework that enables the parallel execution of Python applications in a task-based manner targeting distributed computing platforms. It relies on the COMPSs runtime Badia et al. (2015) to exploit the parallelism of applications by analysing data dependencies between tasks and managing tasks execution.

4.1.1 Programming Model

Using the PyCOMPSs programming model, sequential code can be converted into a task-parallel code by minimal additions of code. In PyCOMPSs, tasks are defined by annotating the methods of the application with Python decorators. Using the @task decorator, a method is identified as a task at execution time. Figure 5 presents a sample PyCOMPSs task. The @task decorator includes information about the return type of this task.

```
@task(returns=int)
def add(a, b):
    return a + b
```

Figure 5. Sample PyCOMPSs Task

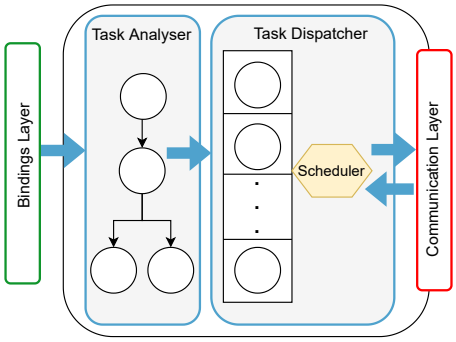
In addition to sequential task execution, PyCOMPSs allows the parallel execution of a task with MPI by means of the @mpi decorator. Tasks decorated with the @mpi decorator are called *Native MPI* tasks Elshazly et al. (2020). Unlike sequential tasks that are executed by a single process, *Native MPI* tasks are executed by MPI processes.

4.1.2 Runtime

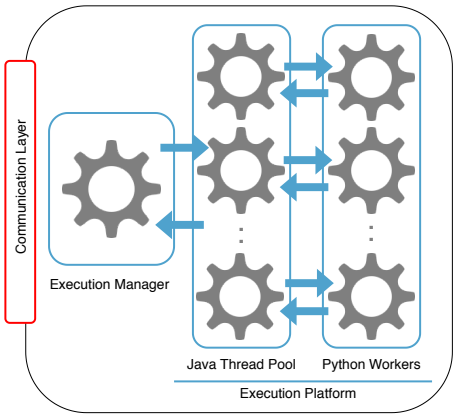
The COMPSs runtime follows a master-worker architecture. At application launch time, the COMPSs runtime deploys a master component on the master node and a worker component on each of the worker nodes. Figure 6 shows an overview of the structure of each component. Task execution calls are then handled by the two main threads of COMPSs master:

- *Task Analyser* which analyses the dependencies between tasks.
- *Task Dispatcher* which consists of a task scheduler that is responsible for scheduling tasks and a submission engine for submitting dependency-free tasks to available resources for execution.

Continuing with Figure 6, Figure 6(b) shows the worker component of COMPSs on each of the worker nodes. Each COMPSs worker consists an *Execution Manager* and an *Execution Platform*. The *Execution Manager* is responsible for launching the execution platform and communicating with COMPSs master through a communication layer. The *Execution Platform* consists of a Java thread pool and a matching number of Python worker processes. Eventually the Python workers are responsible for executing the task. Communication between each Python process and its corresponding Java thread is done through Linux Pipes.



(a) COMPSs Master



(b) COMPSs Worker

Figure 6. COMPSs Runtime Components Overview

A non dependency-free task is blocked until the *Task Dispatcher* receives from the worker node a successful termination signal of its predecessors. When a task execution ends, the executing Python process signals its corresponding Java thread. The Java thread carries out some postmortem operations (such as cleaning the task working directory) and then signals the *Execution Manager* for task execution end. Finally, the *Execution manager* notifies the *Task Dispatcher* to release the successors of that task.

To support tasks execution in a distributed environment, the COMPSs runtime transfers the data of the tasks between the execution nodes in a serialized format. Before starting task execution, the Python process responsible for task execution deserializes the inputs of that task before executing it. Similarly, after the task finishes execution, the worker Python process serializes the returns of that task before propagating its completion message to the components of the system. All the serialization and deserialization details are done in an abstract manner from the user.

4.2 Eager Release of Dependencies In PyCOMPSs

Enabling the eager mechanism for releasing dependencies in PyCOMPSs involved implementation efforts in both components of the PyCOMPSs runtime: the master and worker components. Some parts were re-implemented in the master component where the dependencies are managed and released. Also, a new functionality was implemented in the worker component to enable the triggering of data releases in the user code and notifying the master that an output value is ready.

However, it should be noted that it was not necessary to re-implement the complete PyCOMPSs runtime. For instance, the task scheduler was not re-implemented.

This subsection is divided into two parts. Subsection 4.2.1 describes the adjustments made on dependencies management in the master component to enable parameter-aware dependencies. Subsection 4.2.2 presents an extension to the PyCOMPSs programming model to enable the notification of ready output values and trigger the release of data dependencies before the end of task execution.

4.2.1 Parameter-aware Dependencies

For analysing and scheduling a task, the PyCOMPSs runtime uses two main abstractions:

1. *Parameter*: represents task parameters; its ID, type and whether it is input or output.
2. *Execution Action*: represents a task instance that is ready for scheduling and execution. It contains the predecessor and successor sets of a task.

Whenever the COMPSs master receives a task execution request, the *Task Analyser* first identifies if this task has any data dependency relationships with the previously received tasks.

Once the task analysis has been done, the *Task Dispatcher* instantiates and uses *Execution Action* objects for scheduling, execution and releasing successors when the execution of the predecessor tasks are completed. Every task has a *Predecessor Set* that contains the task IDs of all its predecessors and a *Successor Set* that contains the task IDs of all its successors. Listing 1 shows a pseudo code of the process of building the predecessor set and the successor set of a given *Execution Action*.

Listing 1: Parameter-unaware Dependency Management

```
Predecessors = task.getPredecessors();
for Task p: Predecessors do
    for ExecutionAction e: p.getExecutionActions()
     do
        this.PredecessorSet.add(e);
        e.SuccessorSet.add(this);
    end
end
```

When the predecessor set of a certain successor task becomes empty, this successor is marked as free of dependencies and released for execution. As noted in Listing

1, data dependency parameters are not included in managing the dependency relationship thus making them tasks-only specified relationships.

To enable the parameter-aware specification of dependencies, we used a new abstraction called *Task Dependency Parameter*. This abstraction creates a relationship between *Parameter* objects and the *Execution Action* objects of the predecessor task. Hence, the runtime can easily identify which task produces a certain parameter.

Listing 2 depicts the pseudo code for creating parameter-aware dependency relationships. All the *Execution Action* instances of the predecessor task are updated to contain the ID of the parameter that resulted in the dependency.

Listing 2: Parameter-Aware Dependency Management

```
for Parameter p: task.getParameters() do
    TaskDependencyParameter tdp =
     (TaskDependencyParameter) p;
    predecessor = tdp.getPredecessor();
    if predecessor != Null then
        for ExecutionAction e:
         p.getExecutionActions() do
            this.PredecessorSet.add(p.id, e);
            e.SuccessorSet.add(p.id, this);
        end
    end
end
```

As can be noted in Listing 2, instead of managing dependencies only using the *Execution Action* objects that represent running instances of the tasks, the parameter-aware implementation additionally uses parameters IDs; the sets of predecessors and successors are updated with a pair of the form: <*Parameter ID*, *Execution Action*>.

Consequently, when the *Task Dispatcher* gets notified that a task has generated a data/parameter, the corresponding ID of that parameter will be removed from all successor sets of a task. Hence, this task can be released for execution if it does not require any other data.

4.2.2 Triggering Dependencies Release

We extended the programming model of PyCOMPSs to include a new API to indicate that data has been generated, hence, triggering the dependency release mechanism in the master component of COMPSs without having to wait for the task execution to end. This API has the following form:

compss_ready_value(Data Object, Index)

This notification API requires two parameters as input: Index which is a unique identifier for the generated data that will be mapped to a parameter ID, and Data Object which is the actual data to be returned.

Figure 7 illustrates a sample task code that uses compss_ready_value() to return two outputs at two different times of a task execution.

It should be noted that in Figure 7, the number of output values of the task is specified in the @task decorator. Using this information, the programming model and runtime know exactly how many outputs to expect from a task. On the one hand, the runtime will be able to carry out necessary management operations such as instantiating *Parameter* and *Task Dependency Parameter* objects and creating the corresponding dependency relationships. On the other hand, the programming model will allow iterating and indexing the task outputs in the user code. Otherwise, if the number of returns is not specified, the returns of the tasks will be treated as one single object in the runtime and the programming model.

```
@task(returns=2)
def sample_task():
    # performs lengthy computation on variable a
    compss_ready_value(a, 0)

    .....

    # performs more computation on variable b
    compss_ready_value(b, 1)
```

Figure 7. Sample Task using compss_ready_value to release Output Values.

In order to differentiate between different runtime events, we added a *readyValue* message to indicate the event of generation of a data value. This message is of the following form:

readyValue (TaskID, ParamID)

Whereas to indicate the event of task termination, the runtime uses the *endTask* message in the following form:

endTask (TaskID)

Figure 8 presents a high level sequence diagram of the different trigger messages and how they are handled by the worker several components. Once the Python worker that executes the task receives a compss_ready_value call, it first maps the index passed in the call to a parameter ID and then signals its corresponding Java executor thread with a *readyValue* message. As soon as the Java executor thread receives a *readyValue* message it notifies the *Execution Manager* that a value was received.

Once the execution of the task code finishes, the control flow goes back the Python worker that triggers an *endTask* message to the Java executor thread to indicate that a task execution has finished. When the Java executor receives an *endTask* message, it notifies the *Execution Manager* that a task execution has finished. Both messages eventually are handled by the *Execution Manager* that notifies the COMPSs master that a certain event has occurred.

On one hand, when the *Task Dispatcher* receives a *readyValue* message, it fetches the *Execution Action* object corresponding to the task ID provided in the *readyValue* message. For a given parameter ID, it releases all the execution actions associated with that parameter ID. On the other hand, when the *Task Dispatcher* receives *endTask* message, it releases the remaining successors -if any- and carries out postmortem operations related to the task that finished.

The serialization of the ready values is done as soon as the Python worker receives the compss_ready_value() call and

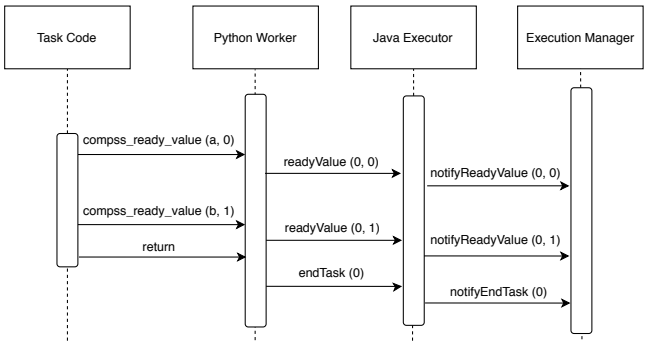


Figure 8. Worker Execution Workflow using compss_ready_value().

not deferred until the task finishes execution. Figure 9 shows a timeline for the different operations carried out in a task releasing two values. Before executing the user code in the task, the python worker process that handles the execution of the task deserializes the inputs of that task so that they can be used inside it. In the lazy approach of releasing dependencies, when the task returns (i.e. the execution of the user code ends), the executing process returns to the Python worker to serialize all the returns of the task. On the other hand, using compss_ready_value() to trigger the release of dependencies, the value is serialized once the call is made. Once the value is serialized, the Python worker process starts the workflow of notifying the runtime that a return value is available.

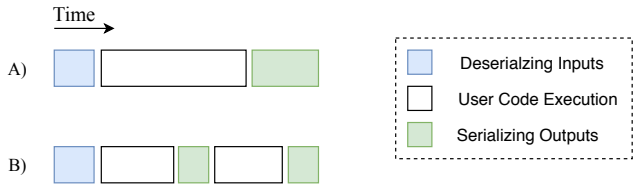


Figure 9. Operations carried out During Task Execution. In a lazy-release of dependencies (A), all the returns are serialized when the task execution ends. Whereas using *compss_ready_value()* (B), return values are serialized once the call is made.

5 Evaluation

In this section, we evaluate our proposal against the default lazy-release of dependencies. In all the experiments, we used the default dependencies release approach of PyCOMPSs as the baseline. We first start by describing the infrastructure in Section 5.1. Section 5.2 presents an evaluation of the overhead associated with using compss_ready_value to eagerly release data dependencies in PyCOMPSs. Finally, Section 5.3 presents the impact of using eager-release of dependencies on the performance of use cases that exhibit real patterns.

5.1 Infrastructure

All experiments were run on the MareNostrum 4 supercomputer, located in the Barcelona Supercomputing Center (BSC) (MareNostrum 4 BSC-CNS, 2017). MareNostrum 4 is composed of 3456 nodes, each node with 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each (at 2.10 GHz) for a total of 48 cores per node. Nodes are interconnected with 100 Gbit/s Intel Omni-Path. Each node is equipped with a SSD disk of 200 GB as local storage and has access to shared disks mounted on General Parallel Filesystem (GPFS).

The MareNostrum 4 supercomputer provides services to researchers from different disciplines such as life sciences, geological sciences and computing architecture.

Due to the master-worker deployment architecture of PyCOMPSs, each submission to the supercomputer queuing system was done with the number of worker nodes plus one that is dedicated as the master node. In all the experiments of this section, we mentioned only the number of worker nodes used for each experiment. In this configuration, the master node only launches and manages the execution on worker nodes and does not perform any computations.

5.2 Overhead Evaluation

This subsection examines three overhead aspects of using an eager-release approach in PyCOMPSs. First, it shows the impact of making compss_ready_value calls to return increasing the number of objects. Then, it presents the impact of increasing the sizes of returned objects. Finally, it examines the network overhead caused by using compss_ready_value calls and its effect on the total time. All results in this subsection were obtained running on a set of 6 nodes of MareNostrum 4, using one node as master node and five nodes as worker nodes.

To this end, we developed a benchmark that has a task graph $G = (p, s_1, s_2, .., s_n)$ where p is the predecessor task and s_i where $i \in n$ are successor tasks. The number of successor tasks is equal to the number of objects returned by the predecessor task such that each successor requires only one output of the predecessor task (i.e. $I(s_i) \cap O(p) = \{d_i\}$). The predecessor/generator task generates and returns objects whereas successor/consumer tasks do not perform any computations. In order to be able to test different aspects of the eager-release mechanism, the generator task can be tuned to return different number of objects or different sizes of objects. All the experiments in this subsection were run two times: one time where the generator eagerly releases dependencies (i.e. eager generator). The second run has a default lazy generator that releases all the dependencies when task execution ends (i.e. lazy generator).

5.2.1 Impact of Increasing The Number of Returned Objects

For measuring the impact of returning different number of objects using compss_ready_value, we ran the benchmark multiple times, each time with different number of returns. In all the runs, all returns are of the same size. The eager generator uses compss_ready_value to return an integer each time it is called, whereas the lazy generator returns all values at the end of task execution.

Figure 10 shows the impact of increasing the number of returned objects on task time and total time with both dependency release approaches. For fewer number of returns (10 and 100), the task time and total time in both approaches are almost the same. As the number of returns increases

in the eager-release approach, the overhead of making the compss_ready_value call starts to appear and the task time increases. However, as task time increases in the eager-release case, the total time decreases for larger number of returns in comparison to the lazy-release case.

The time increase in the task due to the overhead of making compss_ready_value calls is compensated by the fact that using an eager-release approach, tasks are released earlier for execution. The execution of consumer tasks is overlapped with the execution of the generator task so the total time decreases. Whereas in the lazy-release approach, as the number of returns increases, the amount of tasks to be executed after the generator task increases and total time also increases.

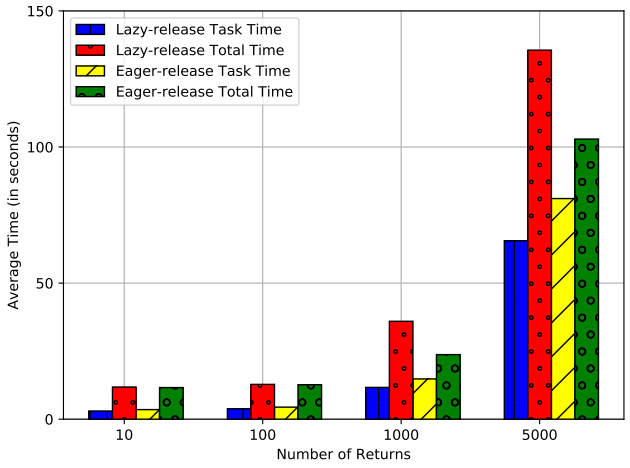


Figure 10. Impact of Increasing Number of Returns on Task and Total Time

5.2.2 Impact of Increasing The Sizes of Returned Objects

Next, we measure the impact of using compss_ready_value to return multiple Python list objects of different sizes. This experiment aims to specifically measure the effect of data serialization on task time and total time. To this end, we fixed the number of returns in both the eager generator and lazy generator to 1000 returns. We launched five runs, the size of the returned lists increases by one million integer in each run.

Figure 11 shows that as the lists sizes increase, no significant difference can be observed on the task time and the total time between the lazy-release approach and the eager-release approach. In addition to that, the gain achieved from using eager-release approach is almost the same for different return sizes. As the sizes of the lists increase, the average time of the generator task increases. However, since the successor tasks are dummy tasks that does not perform any computation, their execution time for different return sizes is almost the same. Thus, the total time increases but the gain remains the same.

Figure 12 offers a closer look at the time spent in serialization in both release approaches. In both cases, as the size of return increases, the serialization time increases. However, in lazy-release approach (Figure 12(a)) the serialization of all 1000 returned lists is carried out after the task code ends. After the Python worker serializes all the

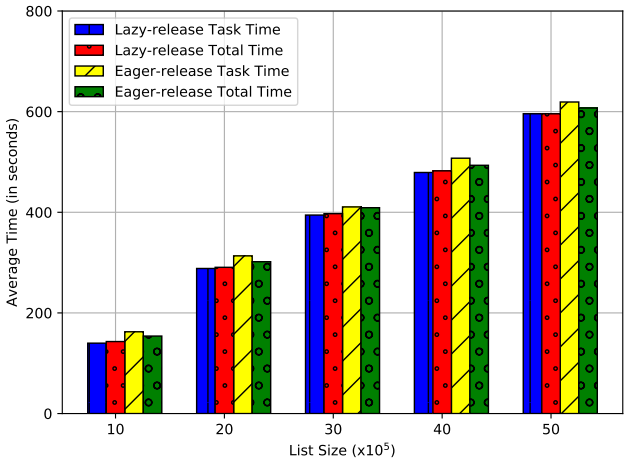
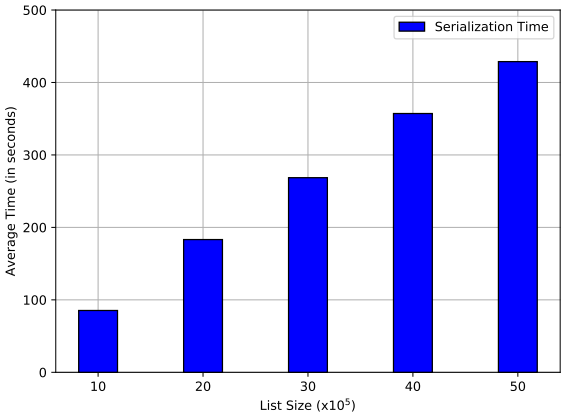
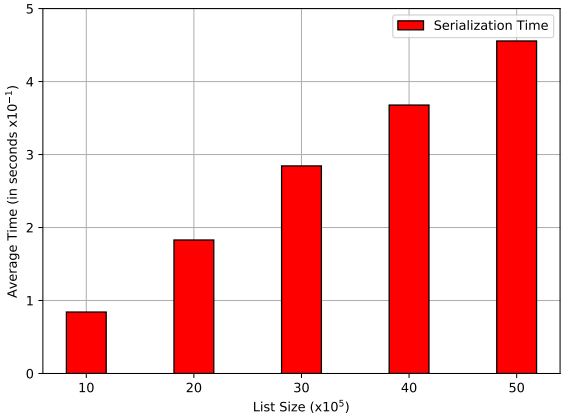


Figure 11. Impact of Increasing Sizes of Returns on Task Time and Total Time

returns, the *endTask* message is triggered. Whereas in eager-release approach using compss_ready_value (Figure 12(b)), the serialization is done only for the return value that is passed in the compss_ready_value call. As soon as this return value is serialized, a *readyValue* message is triggered for that value.



(a) Serialization of All Returns in Python Worker after Task Execution



(b) Serialization of The Return Value in compss_ready_value call

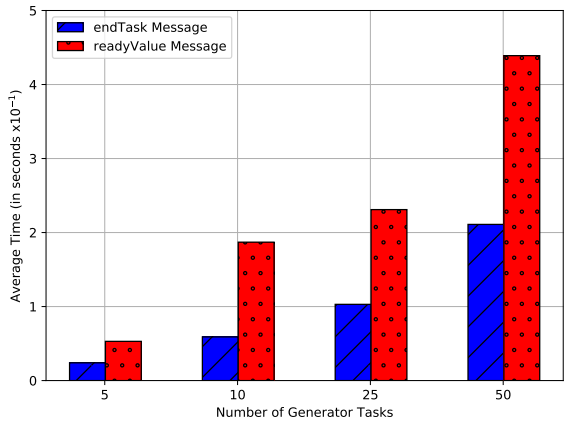
Figure 12. Serialization Time With Increasing Sizes of Returns In Lazy-Release Approach and Eager-Release Approach

5.2.3 Impact of Network Overhead

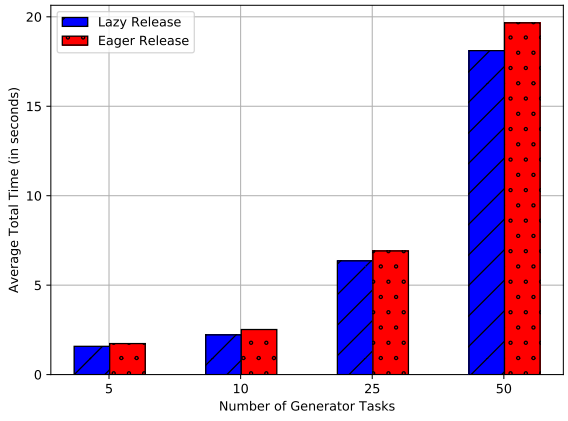
Finally, this experiment measures the network overhead of using an eager-release approach to release dependencies and its impact on the total time. As explained in Subsection 4.2, every time a compss_ready_value is called, it uses a *readyValue* message to notify the runtime that the return value passed in the call has been produced. Whereas the lazy-release approach uses only one *endTask* message after the task ends to notify the availability of all the return values of the task.

Several runs were launched, each run has a different number of generator tasks where each generator returns 100 integers. Figure 13(a) shows that the average time for receiving a *readyValue* message in the eager-release case and an *endTask* message in the lazy-release case increases as the number of generators increase. It can be noted that the average time of receiving *readyValue* message is higher than receiving *endTask* message. As the number of generators increases, the number of returns increases and *readyValue* messages start flooding the network. This network overhead affects the total time as shown in Figure 13(b).

Nevertheless, the effect of network overhead on the total time could be mitigated in real applications where successor tasks spend time performing computation as will be demonstrated in the use cases in the next subsection.



(a) Average Time to Receive Release Messages with Increasing Numbers of Generators



(b) Total Time with Increasing Number of Generators

Figure 13. Impact on Network with Increasing Number of Generators

5.3 Use Cases

In this subsection, we measured the impact of our proposal on the performance of two different use cases. Each use case exhibits a different rate of releasing dependencies, thus, showing different performance aspects of the eager-release mechanism. Use case 5.3.1 features a sequential generator task where data are returned and dependencies are released in a time interleaving manner. Whereas use case 5.3.2 returns data and releases dependencies at a higher rate using a parallel generator task. Finally, Section 5.3.3 presents a use case where the producer task is a streaming task.

5.3.1 Domain Decomposition of Geometrical Shapes

Due to the time and memory required for solving problems in mechanical and engineering disciplines, domain decomposition techniques are applied to split a global domain into sub-domains. Later, a partial differential equation (PDE) is performed on each sub-domain in a parallel manner. In these problems, sub-domains are not of equal dimensions because of the irregular geometry of the domain.

Therefore, this problem is a good candidate to study the impact of eager dependencies release as implemented in PyCOMPSs. We developed an application that decomposes a 2D mesh using an iterative solver into several sub-domains and then it applies a partial differential equation on each sub-domain. It should be noted that this application and its workload pattern corresponds to a real workflow used in the field of mechanical engineering modeling. The iterative solver used for creating sub-domains is called the *FETI Method* Farhat et al. (2001), which is a well-known domain decomposition method in the field of numerical analysis. In addition to that, the pattern that this application mimics is used in the modeling experiments of The EXPERTISE Project, which is a European multidisciplinary project that aims at the modeling of turbine components.

Figure 14 presents the PyCOMPSs task graph skeleton of the application. A domain decomposition generator task returns N sub-domains. Each sub-domain will be processed by one consumer successor task, where each consumer will apply a PDE on its input sub-domain.

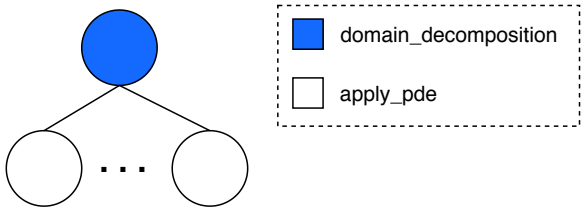


Figure 14. Skeleton Graph of Domain Decomposition Application

We have measured the performance of the application with eager dependency release and lazy dependency release on 8 working nodes of MareNostrum 4. Since the number of sub-domains returned by the generator task has an important effect on the availability of work (and hence performance),

we run the experiments with a variable number of data returns per generator task. In each experiment, we used 4 domain decomposition tasks. Each task decomposes one domain and returns several sub-domains. In addition to that, we maintained the number of consumer tasks equal to the number of sub-domain releases (i.e. in all experiments, one consumer task is used to process one sub-domain).

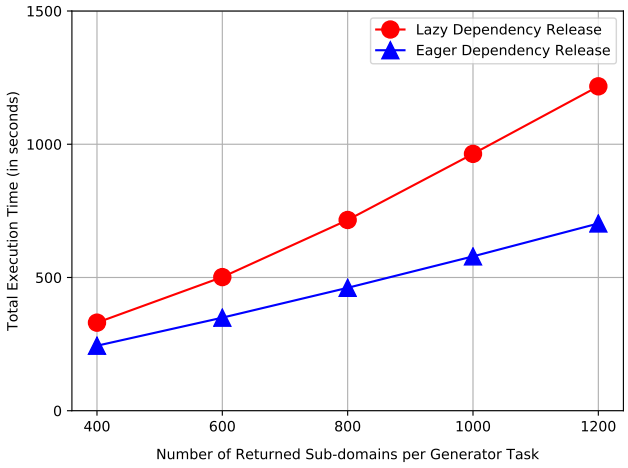
Figure 15 shows the execution time of running the application with eager dependency release and lazy dependency release and the performance gain achieved with increased number of sub-domains. As shown in Figure 15(a), as the number of sub-domains returned per generator task increases, the performance improvement obtained by the eager dependency release mechanism increases. This is because in the lazy-release case, as the number of sub-domains increases, the number of consumer tasks to be executed after the generator task ends increases. In this case, all consumer tasks require execution resources at the same time, thus tasks spend more time waiting for available resources. Whereas in the eager-release case; as soon as the runtime is made aware of a generated sub-domain, it releases the consumer task that has a data dependency with that sub-domain. Hence, consumer tasks that are ready for execution do not spend as much time waiting for execution resources as they do in the lazy-release case when the generator finishes execution.

Figure 15(b) presents the performance gain achieved using eager-release as the number of returned sub-domains increases over using the default lazy-release of PyCOMPSs. As the number of returned sub-domains increases, the execution of the eagerly-released consumer tasks starts earlier and overlaps with the execution of generator task. Therefore, unlike lazy-release case, less consumer tasks are left to be executed after the generator task ends. Hence, tasks spend less time waiting for a free resource.

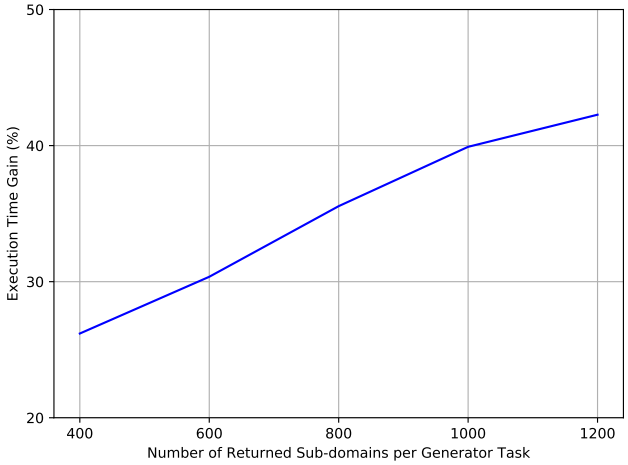
It should be noted that the number of consumer tasks with respect to the number of available resources have an impact on the performance gain. On one hand, as the number of returned sub-domain increases, consumer tasks overwhelm the available resources after the generator task end in the lazy-release case. Consequently, consumer tasks spend more time waiting for free resource and the execution time increases. On the other hand, when the number of returned sub-domains decreases with respect to the number available resources, performance gain of using eager-release decreases over using a lazy-release approach. This is because there are enough resources to execute tasks so consumer tasks will wait less time to be executed.

Figure 16 presents the scalability results of the application with both dependency release approaches. For the strong scalability experiments the number of returns is kept fixed while the amount of worker nodes is increased. On the other hand, for the weak scalability experiments, the data set is increased in the same proportion as the amount of worker nodes.

For the strong scalability experiments, we used 10 domain decomposition generator tasks, each task returning 800 sub-domains. This workload is fixed for all the number of nodes. As for the weak scalability experiment, the number of sub-domains increases from 800 sub-domain to 9600 sub-domain as the number of workers increases. For



(a) Total Execution Time with Increasing Number of Sub-domains



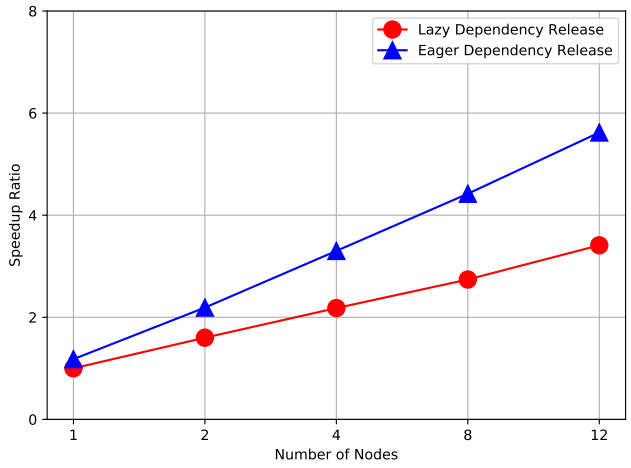
(b) Execution Time Gain with Increasing Number of Sub-domains

Figure 15. Performance Results with Several Number of Sub-domains per Generator Task

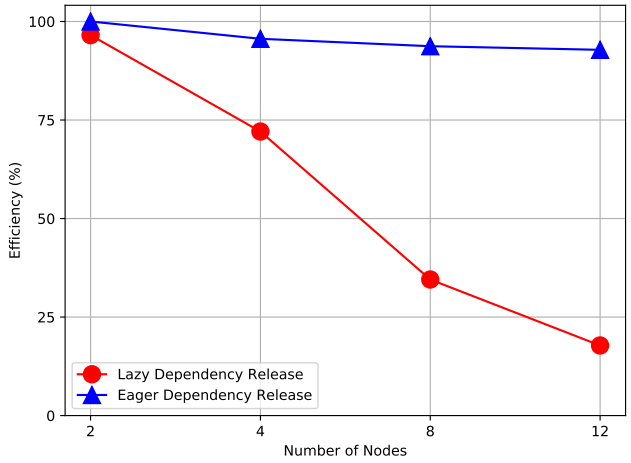
example, for one worker node, we launched one domain decomposition generator task that returns 800 sub-domains. For two workers, we launched two domain decomposition tasks each returns 800 sub-domains and as the number of nodes increased the workload is increased in the same pattern so that each worker would perform the same amount of work.

Figure 16(a) shows that eager-release approach achieves better strong scalability. With fewer number of nodes, the performance improvement of both approaches is somehow close because there is no enough resources to execute ready consumers. However, as the number of nodes increases, the eager-release approach achieves more and more performance improvement since there are more resources to execute ready tasks.

Moreover, in Figure 16(b), unlike the lazy-release approach, an eager-release approach shows a more steady efficiency trend with increasing workloads. As the workload increases in the lazy-release approach, the load on the runtime components increases since all the tasks need to be processed at the same time. However, using an eager-release approach, the workload does not create a bottleneck at any of the runtime components since tasks are handled (and their successors are dispatched) at interleaving time intervals.



(a) Strong Scalability



(b) Weak Scalability

Figure 16. Scalability Results of Domain Decomposition Application

It should be noted that the gain achieved with increasing number of returned sub-domains hides the overhead of compss_ready_value and the network overhead.

5.3.2 Web Archives Analysis

Web archives are stored in a special format called WARC (Web ARChive) Web Archive Format; a file format used for archiving web pages that it is widely used for web crawling and text analysis applications. Each WARC stores several web archives with different sizes.

We developed an application that reads in parallel the records of WARC files and calculates the term frequency (TF-IDF) of each record. Figure 17 shows the PyCOMPSs skeleton graph of the application. A generator Native MPI task reads N records in parallel using MPI. For each record, there are two consuming tasks that will carry out a TF-IDF operation. Since WARC records are of different sizes, MPI processes will spend variable times reading them, hence their successors are released at different times. To run the experiments of this use case, we used datasets from Common Crawl; a nonprofit website that freely provides web archives and web crawl data.

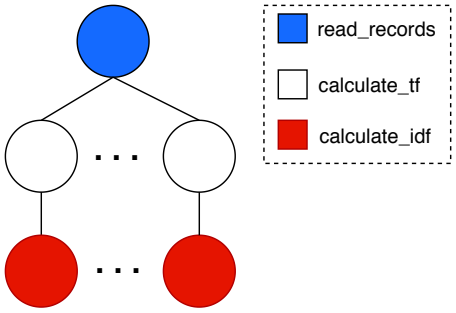


Figure 17. Skeleton Graph of WARC Analysis

For running the experiments of this use case, we used dedicated nodes for doing I/O (called I/O nodes) and dedicated nodes for doing computation (called compute nodes). By using the ProcessorName propriety of the @constraint decorator in the application code, PyCOMPSs schedules each task to matching resources. This application is annotated in such a way that Native MPI I/O tasks will be scheduled to I/O nodes and compute tasks will be scheduled to compute nodes. This way of organizing the infrastructure mitigates the interference levels between I/O and compute tasks which leads to better overall performance as proven in previous I/O research Ali et al. (2009), Liu et al. (2012).

To show the impact of using eager-release of dependencies when using a parallel MPI task, we used 6 I/O nodes and 8 computing nodes. One I/O task launches 288 MPI processes across the I/O nodes. Each MPI process reads a chunk of records from a WARC file that contains 6000 records. The number of records to be read is divided equally across the MPI processes with the MPI process of last rank reading any remainders. For each record, TF-IDF computation tasks are scheduled to the computing nodes. Figure 18 shows the distribution of record sizes of the sample WARC file used in this experiment. Records are of different sizes so MPI processes will take variable amounts of time reading them and their dependencies will be released at different times.

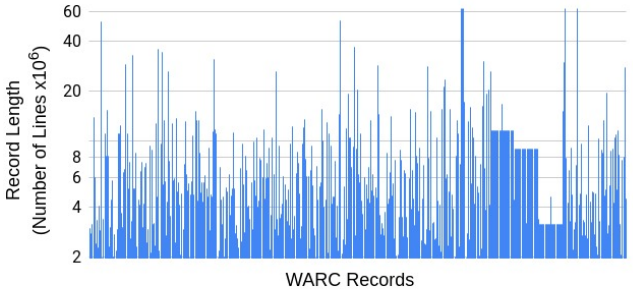


Figure 18. Distribution of Records Lengths in a WARC File

Figure 19 presents the performance results of the application when reading different number of records. Similar to the previous use case, using the eager dependency release achieves performance improvement over the lazy release. In the lazy dependency release, all the consumer

tasks wait until the generator task ends to be scheduled and executed. Once the read task ends, consumer tasks overwhelm the computing resources and hence, most of them spend more time waiting until there is an available execution resource. On the other hand, with eager release of dependencies, consumer computation tasks are released as soon as their data dependency is read and computation overlaps with I/O. Therefore, better performance is achieved as consumer tasks start execution while the read task is still running. Hence, avoiding resource and system contention that happens in the lazy dependency release case when the reading task ends.

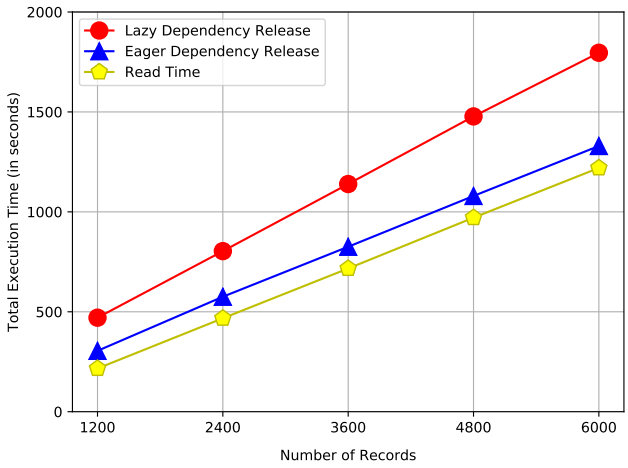


Figure 19. Performance Results with Increasing Number of Records

Moreover, looking closely at Figure 19, the performance gain of the eager-release approach varies between the read time (which is the maximum achievable performance improvement) and the lazy-release time (which is the minimum achievable performance improvement). In both scenarios, the performance of an eager-release will never be better than the reading time and will not be worst than the lazy-release performance.

To better understand the results of Figure 19, we refer to Figure 20 which shows two screenshots of the execution traces of one of the experiments using the lazy dependency release and eager dependency release. To generate execution traces, the runtime of PyCOMPSs uses the Extrae Package to instrument the start and end of each task. At the end of the execution, the generated trace file can be viewed using the Paraver Tool. Both screenshots in Figure 20 has X-axis that represents time and Y-axis that represents execution threads; each thread executes one task at a time. The execution traces in Figure 20 show a blue MPI read task that spans MPI processes across I/O nodes. Each record is processed by a calculate_tf white task then a calculate_idf red task on the compute nodes.

As it appears in Figure 20, in the case of eager-release, records of small and medium sizes can be executed as soon as they are read without having to wait the *read_records* task to read all records. While the MPI processes of the *read_records* task spend more time reading larger records, the eager-release execution releases the dependencies of small and medium size records and starts their processing. On the contrary, in the case of the lazy-release execution, the processing of the small and medium size records is blocked until the *read_records* task reads all the records. After the *read_records* task finishes execution, all dependencies are released and *calculate_tf* and *calculate_idf* tasks overwhelm the execution resources, hence, tasks have to wait for available resources to be start execution.

Since data returns are done in parallel, the amount of performance improvement achieved in the eager-release execution is proportional to how fast the parallel processes will return a value and release its dependency. In the WARC analysis application, the performance improvement depends on the number of small and medium sizes records compared to the number of larger records. As this number increases, more performance improvement will be achieved using eager-release of dependencies. Nevertheless, if all records have the same size, the MPI processes will spend almost equal time reading them and their dependencies will be released at the same time. Consequently, their execution will not overlap with the reading task and the eager-release approach will behave like a lazy-release one.

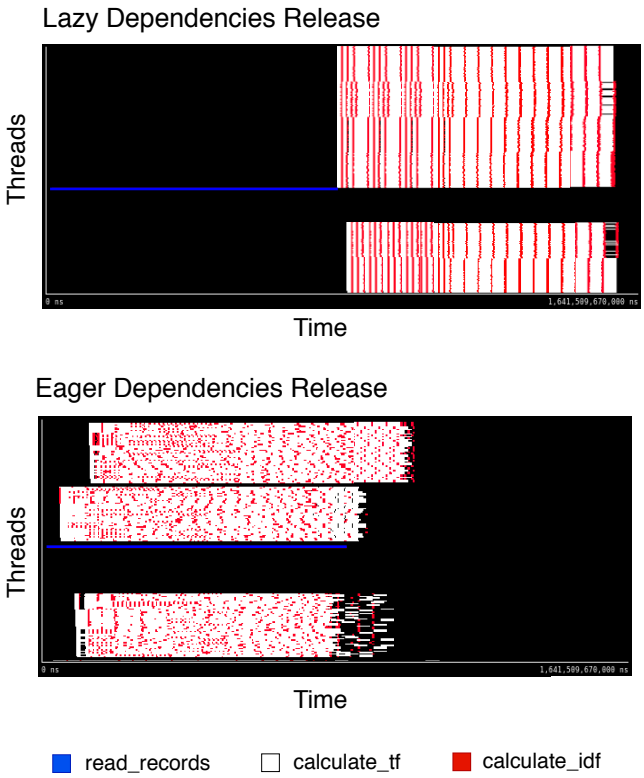


Figure 20. Execution Traces of WARC Analysis Application (Top: trace of the lazy dependency release; bottom: trace of the eager dependency release)

To test the effect of the rate of data returns on the performance improvement using an eager-release approach, we ran the application with several artificial workloads. Each workload has a different distribution of record sizes, hence, MPI processes will spend different times reading them and their dependencies will be released at different rates. Figure 21 shows the performance gain achieved with different ratios between the number of small and medium records to the number of large records. When all records are of the same size, no performance gain is achieved and the eager-release approach behaves like a lazy-release approach. Because there

is no opportunity for computation overlapping. However, when the number of small and medium records increases, performance gain increases because their execution overlaps with the read task that is still reading larger records.

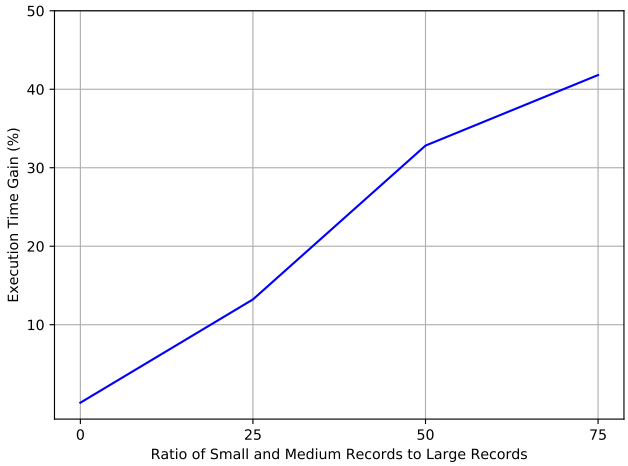


Figure 21. Execution Time Gain with Different Ratio of Small and Medium Records to Large Records

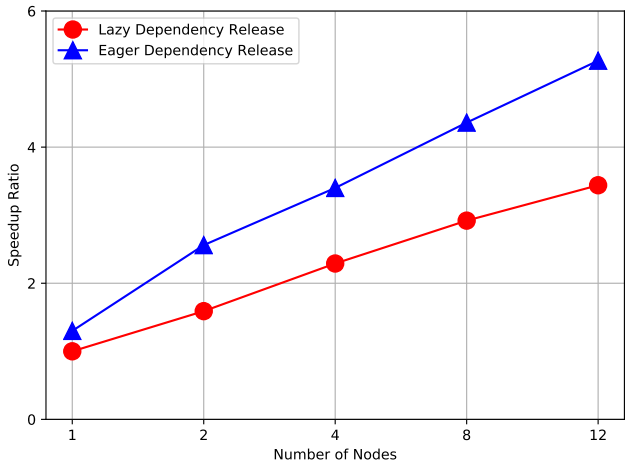
For testing the scalability of using eagerly releasing dependencies in a parallel task, we tested this application with different number of nodes and different workloads. For the scalability experiments we used a WARC file which contains 24000 record of different sizes.

For running the strong scalability experiments, the workload is maintained fixed by using the same number of I/O nodes and increasing the number of compute nodes. Each I/O node runs one I/O native MPI read task that launches 48 MPI process per node for a total number of 240 MPI process across all the I/O nodes. Each MPI process reads 100 record from the WARC file. For each record, TF-IDF compute tasks are launched on any of the compute nodes.

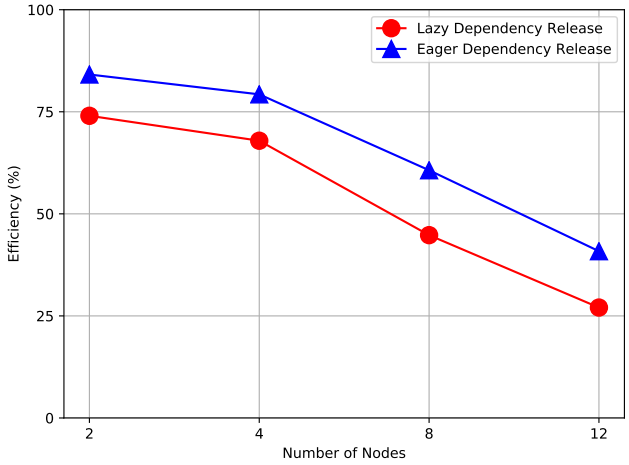
As for the weak scalability tests, the workload was changed by changing the number of I/O nodes in each experiment. By changing the number of I/O nodes, the number of I/O tasks changes and the workload changes because the amount of records to be read changes. In these experiments, each I/O node hosts a parallel MPI read task that launches 48 MPI process per I/O node, each MPI process reads 100 record from the file. In the eager-release case, once MPI process reads a record, it uses compss_ready_value to release the successor compute task.

Figure 22 presents the scalability results of the application. In Figure 22(a), an eager-release approach achieves better strong scalability than the lazy-release approach. With eager-release of dependencies, as the number of computing nodes increases, more resources are available to satisfy the consumer tasks that are eagerly released. Given that the amount of work for each MPI process is different so that dependencies will be released at different times allowing overlapping computation, and thus, performance improvement.

Furthermore, Figure 22(b) illustrates the weak scalability of the application. An eager-release approach also achieves better weak scalability than the lazy-release approach. The more data returned and released in parallel, the more workload the components of the system have to handle. In



(a) Strong Scalability



(b) Weak Scalability

Figure 22. Scalability Results of WARC Analysis Application

addition to that, it should be noted that this use case achieves worst efficiency than the previous use case in Subsection 5.3.1. As the data are returned in parallel, dependencies are released at a higher rate and the overhead of the system increases resulting in a similar trend but better performance compared to the lazy-release case.

5.3.3 *Pairwise Sequence Alignment*

In the fields of life sciences and Bioinformatics, pairwise sequence alignment (or mapping) is used to identify regions of similarity between two DNA or RNA sequences. Sequence alignment is carried out by mapping tools that take two inputs: an unknown (or target) sequence and a known sequence (or a reference) with the goal of identifying whether both sequences have a structural, functional or evolutionary relationship.

Sequence files are usually of big sizes as they contain a large number of DNA or RNA fragments, called *Reads*. Therefore, an out-of-core approach is used to align the target sequence files to the reference file since the whole target sequence file is too big to fit in memory. In this approach, the target file is processed in parts; the application reads a

part that can fit in memory then processes it before reading the next part.

A different solution to this problem is to stream the target sequence file such that the mapper tool processes streams of reads. This way, the cost associated to keeping large number of reads in memory is eliminated. Our proposed approach of eagerly releasing data dependencies makes this solution possible in a task-based execution context; the processing of reads can start as soon as a read or group of reads are received from the I/O stream. Otherwise, using a streaming solution with lazy dependencies release would not be effective as the reads would still need to be kept in memory. Indeed, this solution approach can be generalized to solve any problem that has inputs of sizes bigger than the available memory or any problem that has a workload that exhibits high memory requirements.

We developed a sequence alignment PyCOMPSs application that has the task skeleton depicted by Figure 23; a *get_reads* task loads reads from a sequence file and distributes groups of reads to *align* successor tasks, each calling a mapper tool on its input reads. In order to show the capabilities of the eager approach for releasing dependencies, we compared two versions of the application in the experiments: (i) an out-of-core version in which the application follows the pattern: *get_reads* task reads part of the target file that can fit in memory, then after it finishes loading that part, *align* tasks are released in a default lazy dependencies release fashion. This pattern gets repeated until the whole target file is processed. (ii) a streaming version in which the *get_reads* task opens an I/O stream to load the reads from the target sequence input file, then eagerly releases an *align* task when a read or group of reads has been received from the stream.

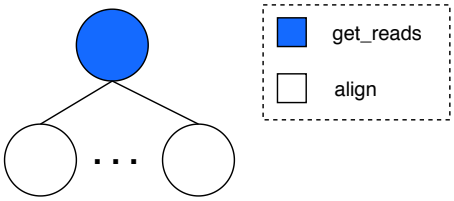
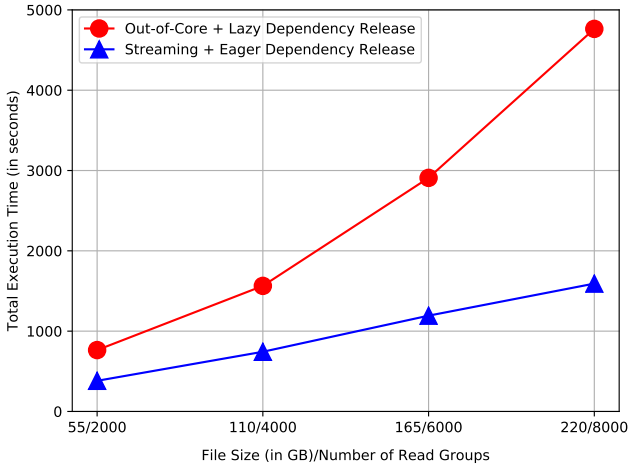


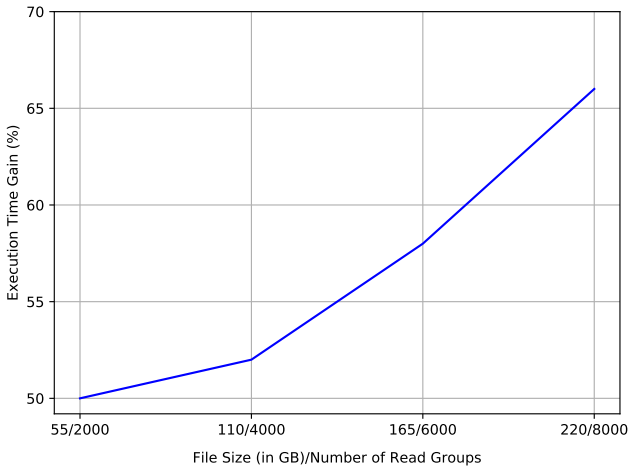
Figure 23. Skeleton Graph of the Pairwise Sequence Alignment Application

For running the experiments, we used different target sequence input files of increasing sizes (55 GB-220 GB). A target sequence file of 55 GB has a total of 685,388,928 reads. No matter what is the size of the target input file, each *align* task processes almost 342,694 reads. The *align* tasks use the popular Burrows-Wheeler Aligner (BWA) Li and Durbin (2010) to align their input group of reads to a subset of the human genome reference (hg38). The input files and the human genome reference are publicly available on the ftp servers of The European Bioinformatics Institute (EMBL-EBI) and the Broad Institute which are well-known resources for providing sequencing data (e.g. sample sequences, genome references, etc..).

Figure 24 shows the significant improvement that can be achieved when streaming the input and eagerly releasing aligning tasks compared to reading parts of the file then releasing all the aligning tasks. As the number of reads is doubled every experiment (file size is doubled), the number of read groups and the number of the *align* tasks that process each of the read groups also doubles. In the case of the out-of-core version, the number of *get_reads* tasks doubles as the file size doubles. For instance, in the case of using an input of 55 GB, one *get_reads* tasks is used to load the whole file in memory, whereas with an input of 110 GB, two *get_reads* tasks are used to read two parts of the file. In order to guarantee the memory requirements of the *get_reads* tasks, we constrained their execution using the *memorySize* argument in the *@constraint* decorator of PyCOMPSs. On the contrary, in the streaming case, only one *read_tasks* is used regardless of the input file size, also without specifying any memory constraints for the task execution.



(a) Total Execution Time with Increasing File Sizes/Number Read Groups



(b) Execution Time Gain with Increasing Number of File Sizes/Number Read Groups

Figure 24. Performance Results with Increasing File Sizes/Number Read Groups

As shown in Figure 24(a), as the target file size and the number of read groups increases, the performance improvement gained by in the case of input streaming and eagerly releasing successor tasks increases. The performance improvement achieved in the case of input streaming and eagerly releasing the *align* tasks is possible because the cost of repeatedly reading parts of the input file in memory is eliminated. The memory requirements of the *get_reads* tasks in the out-of-core version also prevents more *align* tasks to

run in parallel. Moreover, similar to the previous use cases, eagerly releasing the *align* tasks accelerates the execution as tasks execution is overlapped and less tasks overwhelm the infrastructure at the end of *get_reads* task execution. Whereas in the out-of-core and lazy release implementation, all *align* tasks are released after the end of *get_reads* execution. Hence, they overwhelm the infrastructure and each task has to wait more time for computing resources to be available.

Figure 24(b) shows the performance gain in the application. Similar to the domain decomposition use case, described in Subsection 5.3.1, as the file size (the number of read groups) increases, the gain achieved by using input streaming and eager release of dependencies increases.

6 Related Work

In recent years, task-based programming models for executing distributed Python applications have gained popularity. Dask Rocklin (2015) provides a native Python library for creating task-dependency graphs performing operations on NumPy Oliphant (2006) and pandas McKinney (2010) objects. PyCOMPSs, on the other hand, is not limited to specific types of objects. The dependency parameters between PyCOMPSs tasks can be of any type of objects and also files. Dask offers lazy and immediate approaches of execution; in the lazy execution approach, tasks execution does not start unless the user explicitly triggers the computation whereas using the immediate execution, tasks are submitted to Dask scheduler immediately. However, Dask engine identifies dependencies in terms of tasks and releases dependencies only when the execution of the predecessor task completely ends.

Ruffus Goodstadt (2010) and Luigi are programming libraries that offer APIs to create and execute Python applications as workflows. In both libraries, task dependencies are explicitly specified in the user code. Using Luigi, users have to adapt their code to define workflows using the provided object-oriented programming API. In addition to that, task dependencies are hard coded in tasks using an API call. Whereas Ruffus provides an API based on decorators to explicitly specify task dependencies. Moreover, in both Luigi and Ruffus, dependencies are defined in terms of tasks and not dependency parameters. On the other hand, PyCOMPSs is less code-intrusive in the sense that it does not force the users to change or adapt their code to a certain programming paradigm. In addition to that, PyCOMPSs implicitly detects dependencies between tasks which allows for more flexibility and less complexity in application design and programming.

SciLuigi Lampa et al. (2016) offers an extension to the object-oriented programming API of Luigi by separating the dependency definitions from tasks code. unlike Luigi, SciLuigi uses class methods and attributes to define the inputs and outputs of a task (called input and output ports). Then users specify task dependencies by assigning output ports of one task to the input ports of another task, thus declaring a dependency between them. Although SciLuigi follows a dependency-parameter aware approach for defining the dependencies, it does not support the eager release of dependencies. A SciLuigi task has to wait until its predecessor task completely finishes execution even if it does not depend on all of the predecessor outputs. In addition to that, PyCOMPSs offers a more flexible and simpler approach for parallelizing applications. As it does not require the change of user code to fit a specific programming paradigm (i.e. it does not force users to use object oriented programming and it implicitly detects task dependencies).

In non-Python based task-based programming models, Perez et al. (2017) proposed some techniques to improve task nesting and dependencies in OpenMP. One of the proposals, similar in spirit to one of our contributions, includes an API for releasing fine-tuned dependencies in a nested tasks scenario. In this work, an API call enables the release of subtasks without waiting for a super-task to finish its execution.

7 Conclusions and Future Work

This paper presented a proposal to improve the performance of task-based executions. Eagerly releasing data dependencies allows tasks to start execution once their dependencies are ready instead of being delayed until the predecessor tasks finish execution. Using this approach for releasing dependencies enables higher levels of performance by exploiting the inherent parallelism in the applications and overlapping tasks execution.

In this paper, we implemented an eager mechanism for releasing dependencies in PyCOMPSs task-based programming model. This is achieved by introducing two related changes to the design of the system: (1) changing the management of dependencies to be identified not only by tasks, but also by the dependency parameters that caused the dependencies. This approach enables the runtime of PyCOMPSs to release tasks as soon as their data dependencies are available instead of waiting until predecessor tasks finish execution. (2) through the use of compss_ready_value API call, dependencies release can be triggered from the user code as soon as a return value is ready instead of waiting to the end of task execution when the executing process reaches the *return* statement.

This proposal was evaluated against real and simulated workloads and it demonstrated that eagerly releasing dependencies can achieve better performance than using the default lazy approach of releasing dependencies. In addition to that, under certain conditions, the performance gain increases as the workload increases. Using an eager-release mechanism, the execution of tasks overlaps as they are released earlier for execution as opposed to delaying their execution until their predecessor tasks finish execution.

Moreover, the performance gain achieved in an eager-release approach depends on the frequency of output returns. Performance gain is guaranteed when data are returned and dependencies are released in a time interleaving manner allowing overlapping execution. However, when the rate of returns increases, like in the case of a parallel task that returns multiple data dependencies, the performance gain diminishes. As the time between returning values decreases, an eager-release approach follows a similar trend to a lazy-release approach and the overhead of the system starts to affect the performance.

As future work, we plan to investigate approaches to automatically detect the generation of data instead of manually using an API call in the task code. In addition

to that, we plan to extend the eager-release mechanism to support releasing dependencies of nested tasks.

8 Declaration of Conflicting Interests

The author(s) declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

9 Acknowledgements

This work is partially supported by the European Union through the Horizon 2020 research and innovation programme under contracts 721865 (EXPERTISE Project) by the Spanish Government (SEV2015-0493,TIN2015-65316-P) and the Generalitat de Catalunya (contract 2014-SGR-1051).

References

Ali N, Carns P, Iskra K, Kimpe D, Lang S, Latham R, Ross R, Ward L and Sadayappan P (2009) *Scalable I/O forwarding framework for high-performance computing systems*. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. pp. 1–10. DOI:10.1109/CLUSTR.2009.5289188.

Badia RM, Conjero J and Diaz C (2015) *COMP Superscalar, an Interoperable Programming Framework*. *SoftwareX* 3(7-8): 32–36. [Online]. Available:https://doi.org/10.1016/j.softx.2015.10.004.

Broad Institute (2020) Web Page at https://www.broadinstitute.org/. Date of Last Access: 26th October, 2020.

Common Crawl (2007) *Common Crawl Wesbite*. at https://commoncrawl.org/. Date of Last Access: 10th January, 2020.

Dagum L and Menon R (1998) *OpenMP: an industry standard API for shared-memory programming*. *IEEE computational science and engineering* 5(1): 46–55.

Dean J and Ghemawat S (2008) *MapReduce: simplified data processing on large clusters*. *Communications of the ACM* 51(1): 107–113.

Elshazly H, Lordan F, Ejarque J and Badia RM (2020) *Performance Meets Programmability: Enabling Native Python MPI In PyCOMPSs*. In: *28th Euromicro International Conference on Parallel, Distributed and Network-based Processing*.

Extrae Package (2007) *Extrae Wesbite*. at https://tools.bsc.es/paraver. Date of Last Access: 07th February, 2020.

Farhat C, Lesoinne M, Rixen D and et al (2001) Feti-dp: a dual-primal unified feti method - part 1: a faster alternative to the two-level feti method. *International Journal for Numerical Methods in Engineering* 50: 1523–1544.

Goodstadt L (2010) Ruffus: a lightweight Python library for computational pipelines. *Bioinformatics* 26(21): 2778–2779. URL https://doi.org/10.1093/bioinformatics/btq524.

Gropp W, Lusk E and Skjellum A (1999) Using MPI: portable parallel programming with the message-passing interface, volume 1. MIT press.

Lampa S, Alvarsson J and Spjuth O (2016) *Towards agile large-scale predictive modelling in drug discovery with flow-based programming design principles*. *Journal of Cheminformatics* 8.

Li H and Durbin R (2010) Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26(5): 589–595. DOI:10.1093/bioinformatics/btp698. URL https://doi.org/10.1093/bioinformatics/btp698.

Lin-Wen Lee, Scheuermann P and Vingralek R (2000) *File assignment in parallel I/O systems with minimal variance of service time*. *IEEE Transactions on Computers* 49(2): 127–140. DOI:10.1109/12.833109.

Liu N, Cope J, Carns P, Carothers C, Ross R, Grider G, Crume A and Maltzahn C (2012) *On The Role of Burst Buffers in Leadership-class Storage Systems*. In: *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. pp. 1–11. DOI:10.1109/MSST.2012.6232369.

Luigi (2015) *Luigi source code on Github*. Github at https://github.com/spotify/luigi. Date of Last Access: 6th February, 2020.

MareNostrum 4 BSC-CNS, 2017 (2017) *MareNostrum 4 Supercomputer*. Web Page at https://www.bsc.es/marenostrum/marenostrum. Date of Last Access: 26th March, 2020.

McKinney W (2010) *Data Structures for Statistical Computing in Python*. In: van der Walt S and Millman J (eds.) *Proceedings of the 9th Python in Science Conference*. pp. 51 – 56.

Oliphant T (2006) *A guide to NumPy*, volume 1. Trelgol Publishing USA.

Paraver Tool (2007) *Paraver Wesbite*. at https://tools.bsc.es/extrae. Date of Last Access: 07th February, 2020.

Perez J, Beltran V, Labarta J and Ayguadé E (2017) *Improving the Integration of Task Nesting and Dependencies in OpenMP*. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. pp. 809–818. DOI:10.1109/IPDPS.2017.69.

Rocklin M (2015) *Dask: Parallel Computation with Blocked algorithms and Task Scheduling*. In: Huff K and Bergstra J (eds.) *Proceedings of the 14th Python in Science Conference*. pp. 130 – 136.

Tan Z, Zhou W, Feng D and Zhang W (2013) *ALDM: Adaptive Loading Data Migration in Distributed File Systems*. *IEEE Transactions on Magnetics* 49(6): 2645–2652. DOI:10.1109/TMAG.2013.2251616.

Tejedor E, Becerra Y, Alomar G, Queralt A, Badia R, Torres J, Cortes T and Labarta J (2015) *PyCOMPSs: Parallel computational workflows in Python*. *International Journal of High Performance Computing Applications* .

The European Bioinformatics Institute (*EMBL-EBI*) Web Page at http://www.msca-expertise.eu/. Date of Last Access: 26th October, 2020.

The EXPERTISE Project (*EXPERTISE*) Web Page at http://www.msca-expertise.eu/. Date of Last Access: 26th October, 2020.

Web Archive Format (WARC) Web page at http://bibnum.bnf.fr/WARC/. Date of last access: 10th February, 2020.

Zaharia M, Reynold X and et al PW (2016) Apache spark: A unified engine for big data processing. *Communications of the ACM* 59: 56–65. DOI:10.1145/2934664.