Orchestration Framework for Edge Computing with Energy-Aware Scheduling

Abstract—**Edge computing has emerged as a crucial solution to meet the demands of evolving Internet of Things (IoT) applications, which impose strict requirements on latency, reliability, and scalability. While edge computing brings computation and storage closer to data sources, addressing energy consumption remains a significant challenge. This paper proposes an energy-efficient scheduling method for Oakestra, a recently released orchestration framework targeting edge computing environments. Oakestra's hierarchical orchestration effectively manages geo-distributed edge devices but lacks energy efficiency in its scheduling method.**

The proposed energy-efficient scheduling method leverages slack time between microservices to decrease energy consumption through Dynamic Voltage and Frequency Scaling by adapting a variation of the DURPS algorithm within Oakestra. Experiments compare the scheduling length and energy consumption of Oakestra with its original scheduling, and the energy-efficient scheduling method. Detailed analyses are provided for DAG applications, illustrating that the proposed method outperforms the original Oakestra scheduling in terms of energy consumption by trading off an acceptable level of scheduling length increase. The results show an increase in scheduling length between 2-9% and a decrease in energy consumption between 15-43% with the energy-efficient scheduling method, which for edge computing cases could be considered an attractive tradeoff.

Index Terms—**Edge Computing, Virtualization, Energy-Efficiency, Scheduling**

I. Introduction

Edge computing has become increasingly popular as a way to address the needs of emerging Internet of Things (IoT) applications. These applications have stringent requirements on latency, reliability, and scalability and edge computing places computation, storage, and other services closer to data source to achieve better results. In addition, the emerging IoT applications consist of a large number of devices which interact with each other to provide distributed intelligence. [1] However, research has shown that the energy consumption of edge devices is high, and the energy efficiency is poor. This is due to the proliferation of billions of edge devices as "the overall consumption could be significant due to the scale of servers and nodes". [2] Substantial energy consumption also results from the requirements of computation-intensive applications on edge devices. [3] Furthermore, the composition of edge devices is much more diverse than traditional cloud servers i.e. edge devices have different runtimes and handle different data. This makes resource scheduling with high energy efficiency in edge devices a hot topic.

Virtualization technologies provide an abstraction to enable collaboration among heterogeneous devices. However, the existing state-of-the-art orchestration frameworks (e.g. Kubernetes, Docker) perform poorly at the edge since they are designed for reliable, low latency, high bandwidth cloud environments. [4] In addition, the edge devices usually have limited power and resources, which makes them not a good fit for the traditional orchestration framework.

In this work, we will apply a energy-efficient scheduling method into Oakestra [4], a recently released orchestration framework designed for edge computing environment, to make this framework more energy efficient. Oakestra is a hierarchical infrastructure to better manage the geo-distributed and heterogeneous edge devices. It has a root orchestrator which has coarse-grained control over its resources and cluster orchestrators which have the find-grained control of their resources. However, the scheduling method used in Oakestra is not energy efficient enough. As we know that many edge devices have limited power, so we want to involve this factor into the scheduling of this framework, to make it not only designed for geo-distributed and heterogeneous edge devices, but also designed for power-limited edge devices. Our energy-efficient scheduling method is a combination of heterogeneous Earliest-Finish-Time algorithm (HEFT) [25] and the adjusted upward and downward proportionally reclaiming slack algorithm algorithm (DUPRS) [7], where HEFT is a task scheduling algorithm with low time complexity and the DUPRS is a method that takes use of the slack times between microservices to reduce energy consumption through dynamic voltage and frequency scaling (DVFS).

We form a edge computing environment with geo-distributed devices. Our simulation results shows that our energy-efficient scheduling method reduces energy consumption by 16%-45% for three types of applications, while increasing the scheduling length by 3%-7% compared with that of the state-of-the-art. Therefore, Oakestra with energy-efficient scheduling method outperforms the state-of-the-art for geo-distributed edge computing environments in energy consumption with an attractive tradeoff in scheduling length.

This paper mainly has the following contributions:

- We apply an energy-efficient scheduling method to Oakestra, an orchestration framework designed for edge computing environment.
- DVFS is an energy-saving feature of many modern processors to save energy. For the feature to kick-in, the operating system relies on slack time between compute-heavy tasks to reduce the voltage and frequency. Our energy-efficient scheduling method optimizes the slack time between microservices to reduce the energy consumption through DVFS.

- The energy-efficient scheduling method is effective.

The rest of the paper is organized as the follows. Section II outlines related researches. Section III introduces our motivation and problem definition for research objectives. Section IV proposes the energy-efficient scheduling algorithm. The experiments, results and analysis are given in Section V. The conclusion and future work are made in Section VI.

II. Background Research

A. Virtualization

As mentioned earlier, we believe that while there are abundant virtualization technologies out there (i.e. Kubernetes, Docker), they do not perform as well on edge or IoT devices. [4] To this end, we have researched new, state of the art virtualization technologies that are better suited for edge/IoT. "Oakestra" [4] is a recently release orchestration framework that is hierarchical, lightweight, flexible, and scalable, and is designed specifically for edge computing. This framework claims to reduce resource usage by a factor of ten as well as improve application performance to a similar degree. We do acknowledge that Oakestra is relatively new and not as established or well-known in comparison to other orchestration frameworks like Kubernetes, K3s, or KubeFed; however, the reduced resource usage makes it more applicable to edge scenaiors. Other frameworks are not designed for the edge but rather "inherently designed to perform well in managed datacenter networks". Furthermore, they make assumptions about reliability and reachability of devices that may "not necessarily hold at the edge where resources are more dispersed" [4]. According to the authors, unlike Oakestra, these other frameworks also "incorporate many heavyweight operations" in their core components which leads to "[limitations of] their use on constrained hardware" [4]. Lastly, the authors contend that "none of the existing solutions can currently support the edge's heterogeneity in hardware, networking, and resource availability" [4]. For these reasons, we believe that while Oakestra is new and novel, it takes many nuances of edge and IoT compute into its design that were not a central focus in other orchestration frameworks. However, in [4] there is no mention of energy efficiency - which is immensely important to edge computation. Energy consumption of edge devices is an important aspect to optimize, which is why in this paper we augment Oakestra to be more energy aware, and detail our approach in subsequent sections of the paper. Another paper that we have considered in the virtualization category is one that proposes a novel distributed and lightweight virtualization model targeting the edge tier, encompassing the specific requirements of IoT applications [5]. This paper is applicable to our research because this new virtualization model saves energy and reduces traffic in the IoT-Edge infrastructure. Lastly, we researched [6] which introduced a local scheduling algorithm using virtualization. This outperformed the round-robin scheme in terms of the cumulative throughput and response time in the edge computing environment. This is relevant to our research since we want to optimize task scheduling so that edge nodes in order to decrease throughput of traffic between edge nodes.

B. Energy Efficiency

Since our goal is to improve Oakestra and make it more energy aware, we had to survey the field for different methods of achieving energy efficient computation on edge devices. [7] proposes an energy-efficient scheduling with ant colony techniques named downward-and-upward proportionally reclaiming slack algorithm (DURPS) to cut down energy overhead [7] We utilized some components of the DUPRS algorithms in order to efficiently schedule tasks. We utilized the version of the algorithm in [8], since it has the correct latest completion time algorithm in comparison to [7].

Dynamic voltage and frequency scaling (DVFS) is an important methodology used to reduce energy consumption of IoT devices. [9] proposes an application-deadline-aware data offloading scheme using deep reinforcement learning and dynamic voltage and frequency scaling (DVFS) in an edge computing environment to reduce energy consumption. The method reduces the energy consumption by learning the optimal data distribution policies and local computation frequency scaling through interacting with the system and devices. We use DVFS in our Raspberry Pi devices to adjust the frequency of processors, since DVFS can lead to substantial improvements in energy efficiency. [9], [10] and [11] expand further on this topic. There are certainly other methods of energy efficiency. Some papers deal with energy-aware resource scheduling [11], energy-aware placement of edge nodes [12], energy-efficient scheduling based on traffic mapping [3], latency and energy consumption trade offs [13], using power management to reduce energy consumption [14], minimizing energy consumption in vehicular edge computing [10], and some general papers that we used in our research to understand the field of energy consumption in edge/IoT devices [15] [16].

C. Resource Management

In order to achieve energy efficiency, the details of how tasks are scheduled on edge devices across time is paramount. A key paper [17] proposes two methods for task scheduling, the Heterogeneous Earliest-Finish-Time algorithm (HEFT) and the Critical-Path-on-a-Processor (CPOP) algorithm. HEFT is used as a baseline for task scheduling across the domain and we too use HEFT to measure baseline performance in our experiments. Furthermore, we have replaced the preliminary part of DUPRS algorithm in the ant colony paper with HEFT. This is described in more detail in Section IV.

Additionally, there is a broad range of topics related to resource management and energy efficiency. Some papers considered latency and its relationship with energy efficiency. For example in [18] the research tries to minimize the energy consumption under latency constraints and determine whether task execution should happen locally or not. Other papers such as [19] try to determine if certain tasks of an application can be offloaded to the Cloud while remaining tasks can be mapped to heterogeneous cores on mobile devices. This paper had the objective to minimize the total consumed energy under the constraints of task-precedence requirements and application completion time. We also evaluated other areas of research that considered multi-tier edge computing [20] (cloud, backend edge, and frontend edge) in order to minimize task execution cost. Furthermore, [1] introduces distributed resource scheduling methods in edge computing. This paper connects to our hypothesis because it evaluates how distributed resource scheduling can help with computation offloading, bandwidth allocation, power control, and more. This is in addition to general papers on energy-efficient task scheduling that we have considered to get more domain knowledge [21].

III. Sytem Design and Architecture

Our energy-efficient scheduler is a combination of heterogeneous Earliest-Finish-Time algorithm (HEFT) [25] and the adjusted upward and downward proportionally reclaiming slack algorithm algorithm (DUPRS) [7]. The reasons we combine these two algorithms are as the follows:

- The HEFT is a robust method for scheduling tasks in heterogeneous environment effectively. It can generate a preliminary schedule tasks very quickly.
- DVFS is a technique that is normally used in modern processors, which is widely exploited for saving energy through decreasing the supply voltage and frequency of computing units [7]. We want to take a good advantage of this function to save energy consumption of edge devices. DUPRS is an algorithm that uses DVFS to save energy consumption. With the preliminary schedule generated by HEFT, DUPRS can shorten the slack time between tasks by adjusting frequency of processors to save energy.

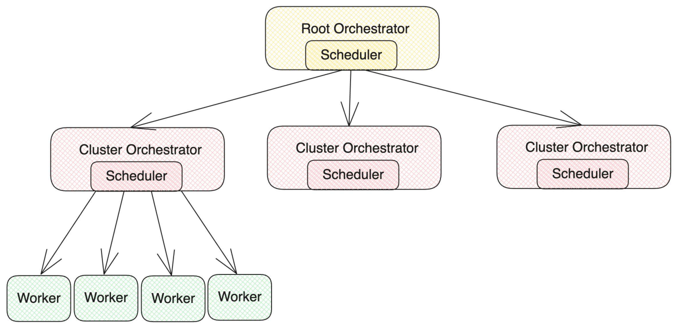

Fig. 1: Oakestra Architecture

A. Oakestra System Architecture

Since our system is an adaptation of Oakestra [4], in this section we provide a short overview of its design. Oakestra is a hierarchical orchestration framework for enabling running edge computing applications on geo-distributed and heterogeneous resources (shown in the Figure 1). Different from the flat management of the state-of-the-art (e.g. Kubernetes) algorithms, Oakestra organizes the infrastructure into distinct clusters in which cluster orchestrators have the fine-grained control of their resources, while the root orchestrator has only coarse-grained control over the global infrastructure. This control separation will make the system more stable and cut down the scheduling time and request execution time. Oakestra comprises of three main entities – root orchestrator, cluster orchestrator, and worker.

The **root orchestrator** is the Oakestra's centralized control plane. The root orchestrator needs to be a machine that can be reachable from all the clusters so that when an infrastructure provider wants to join the system, it can register its resources through the root orchestrator. The root orchestrator has two main responsibilities:

- When an application request comes in, it does the coarse scheduling based on the application's operational requirements and constraints such as hardware constraints and geolocation constraints. These requirements and constraints come with the application request in the SLA descriptor.
- It registers new clusters and collects the information about the clusters' resources.

The **cluster orchestrator** is a logical twin of the root orchestrator. Its management responsibility is restricted to the resources in the cluster. The cluster orchestrator also needs to be a machine that can be reachable from all the workers in the cluster. It has two main responsibilities:

- It needs to periodically updates the root orchestrator with the aggregated statistics of overall cluster utilization and the health of the deployed applications.
- It schedules the tasks in the deployed application to the workers in the clusters based on the tasks' requirements and workers' availability, utilization, and capability.

The **worker nodes** are the edge servers in the clusters. They are the entities responsible for executing the tasks. Different worker has different capacities and capabilities. Each worker needs to periodically updates its cluster orchestrator with its utilization and the health of the tasks.

B. Problem Definition

Oakestra follows a two-step delegated scheduling mechanism. The scheduler at the root orchestrator does the coarse-grained scheduling, while the scheduler at the cluster orchestrator does the fine-grained scheduling.

The root scheduler matches the SLA constraints to the current capacity of each cluster and calculates a priority list of best-fit clusters based on the clusters' latest aggregated statistics. It then offloads the deployment request iteratively to clusters in the list with decreasing priority. The scheduling approach the root scheduler uses is Latency & Distance Aware Placement (LDP). It is an algorithm built based on Resource-Only Match (ROM) algorithm. ROM can find the first resource or the best resource that satisfies the service's capacity requirements. Users can select the first resource or the best resource to find before using the method. Based on ROM, LDP filters unsuitable clusters by comparing their constraints along with approximate geographical operation zones in the SLA requirements.

Oakestra's cluster schedulers also use LDP to schedule tasks. However, this method just considers the applications' capacity requirements, latency and geographical distance constraints. It does not consider the energy consumption as a factor. As we know, many edge devices have power limitation, so it is important to take the energy consumption into the scheduling. In addition, for a cluster, it might not be necessary to consider the geographical constraint because the workers are supposed to be located very close to each other compared with the distance between clusters. Therefore, we want to adjust the scheduling method in clusters to enforce the system to be energy aware. Our scheduling method for clusters combines the method of HEFT [25] and DUPRS [7]. So the problem can be formulated as the follows:

$$\begin{aligned} &minE(A) \\ &s.t.SL_{actual}(A) \leq SL_{con}(A) \end{aligned} \tag{1}$$

$E(A)$ is the energy consumption of application A. $SL_{actual}(A)$ is the actual scheduling length (execution time) of application A. $SL_{con}(A)$ is the scheduling length constraint set for application A ahead of time, which can be the latency requirement of the applications provided by users.

IV. Proposed Algorithm and System Design

We are focusing on the real-time applications which can be decomposed into a set of small tasks and represented by a directed acyclic graph (DAG). The reasons we are paying attention to DAG applications are:

- Modern applications can usually be divided into multiple small tasks and then modeled by a DAG according to the dependencies between tasks, such as chain query, video processing, and data analysis.
- These small tasks are a good fit for edge devices with limited resources.

A DAG application can be written as $G =< N, E >$ where $N = n_1, n_2, ..., n_r$ is the task set and $E \subseteq N * N$ indicates the set of edges. If edge $(n_i, n_j) = 1$, it represents that n_j cannot be executed until n_i is finished, and if edge $(n_i, n_j) = 0$, it represents that n_j does not need wait for the completion of n_i. Fig.2 shows a simple example of a DAG application. For a DAG application, it usually has an entry task which has no predecessor and an exit task which has no successor.

Let $T(n_i, p_k)$ denotes the execution time for n_i executed on processor p_k. Table.I shows the execution time of different tasks executed on different processors at the processors' maximum frequency. The data in Table.I are saved in the clusters for scheduling which can be used repeatedly as long as the same application requests are received. In addition, since the processors/edge devices are usually with limited resources, we assume that a processor can only process one microservice at a time.

A. HEFT

From Fig. 2, we can find that some of the tasks are at the same level and can be executed at the same time, so we need to schedule these tasks at different processors and execute them at the same time to save the total execution time of the whole application. But how to decide when to execute which task? We use the HEFT algorithm to schedule the tasks. After HEFT algorithm, we will have the start time to execute each task on which processor. The reasons we choose HEFT algorithm instead of other more advanced methods are:

- The HEFT is a robust method for scheduling tasks in heterogeneous environment effectively.
- This is just the preliminary step for the scheduling. The result of this step will not be the finalized result.
- Our goal is to minimize the energy consumption, not minimize the execution time. Even though we choose more advanced method, we will still need to adjust the preliminary scheduling to achieve our goal.
- The other advanced methods are more complex than the HEFT algorithm. It is not worth to use a more complex method just to do the preliminary scheduling.

Before we start scheduling the tasks, we need to generate the task priority list based on the dependencies of the tasks. The priority of each task can be calculated as the follows:

$$\begin{aligned} &Priority(n_i) = w_i + max_{n_j \in succ(n_i)} Priority(n_j) \\ &w_i = avgT(n_i, p_k) \end{aligned} \tag{2}$$

w_i is the average execution time for n_i on all the processors. n_j is the intermediate successor of n_i. For the exit task, its

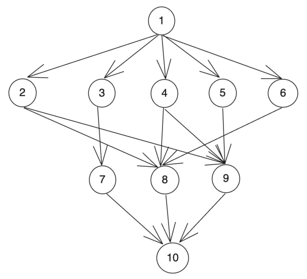

Fig. 2: a DAG application

Task No	p1	p2	p3
1	7	14	25
2	16	19	14
3	11	20	24
4	18	12	20
5	9	20	13
6	11	20	23
7	14	18	8
8	16	19	21
9	11	17	20
10	13	14	11

TABLE I: Execution time for different tasks on different processors

priority is

$$Priority(n_{exit}) = w_{exit} \tag{3}$$

For the case show in Fig.2 and Table.I, we can get a priority list {1, 6, 4, 3, 5, 2, 8, 9, 7, 10}, with task 1 has the highest priority and task 10 as the lowest priority. The priority list should also be saved in the clusters for scheduling the same application requests.

After we get the task priority list, we can assign each task in the task priority list to an appropriate processor one by one. We can first calculate the estimated start time (EST) and estimated finish time (EFT) of each task by:

$$\begin{aligned} &EST(n_{entry}) = 0 \\ &EST(n_i, p_k) \\ &= max\{avail[k], max_{n_m \in pred(n_i)}(EFT(n_m) + c_{m,i})\} \\ &EFT(n_i, p_k) = T(n_i, p_k) + EST(n_i, p_k) \end{aligned} \tag{4}$$

$avail[k]$ is the earliest available time for p_k. n_m is the intermediate predecessor of n_i. $c_{m,i}$ is the communication cost between n_m and n_i. If n_m and n_i are executed on the same processor, $c_{m,i}$ is 0.

$$SL_{actual}(A) = EFT(n_{exit}, p_k) \tag{5}$$

$EFT(n_{exit}, p_k)$ is the estimated finish time of exit task which is also the estimated finish time of the whole application. The purpose of the HEFT algorithm is to minimize $SL_{actual}(A)$. Algorithm 1 is for the HEFT algorithm.

```
Data: task priority list $qlist$, $T(n_i, p_k)$, $c_{m,i}$
Result: $n_i$ is assigned to $p_j$, $EST(n_i, p_j)$,
        $EFT(n_i, p_j)$, $SL_{actual}(A)$
while $qlist \neq \emptyset$ do
    $n_i$ = qlist.pop()
    $j \leftarrow 0$
    $EFT(n_i, p_j) \leftarrow \infty$
    $EST(n_i, p_j) \leftarrow 0$
    $j \leftarrow 0$
    for $p_k \in P$ do
        compute $EST(n_i, p_k)$, $EFT(n_i, p_k)$ (4)
        if $EFT(n_i, p_k) < EFT(n_i, p_j)$ then
            $j \leftarrow k$
            $EFT(n_i, p_j) \leftarrow EFT(n_i, p_k)$
            $EST(n_i, p_j) \leftarrow EST(n_i, p_k)$
        end
    end
end
$SL_{actual}(A) \leftarrow EFT(n_{exit}, p_j)$
```

Algorithm 1: The HEFT algorithm

The complexity of algorithm 1 is $O(E * P)$, where E is the number of edges in the DAG, and P is the number of processors in the system.

B. DUPRS

After the preliminary scheduling for the tasks, we have each task assigned to a specific processor, the estimated start time and estimated finish time of each task, and the execution time of the whole application.

HEFT algorithm is used to find the minimum execution time of the application $SL_{actual}(A)$. However, our goal is to minimize the energy consumption, so we want to trade off some execution time to achieve our goal, as long as the execution time is smaller than the constraint.

We use the DUPRS algorithm to adjust the execution time of a task by adjusting the frequency of the processor that execute the task and to achieve the minimum energy consumption. The reasons why we choose DUPRS are:

- DUPRS is a newly released method focusing on DAG applications, which are also our focus.
- DUPRS is a method using DVFS. Currently a lot of processors are designed to be DVFS to save energy. We think we can take a good use of this characteristics of the processors to reduce the energy consumption. In addition, the edge devices we are using, the Raspberry Pi's, have the function of DVFS.
- DUPRS's objective is to minimize the energy consumption under the latency constraints which is the same as our objective.

DUPRS is a combination of two parts, part 1 and part 2. Part 1 is shortening the slack time between tasks from the head of the priority list while part 2 is shortening the slack time between tasks from the end of the priority list. These two parts both use DVFS, adjusting the frequency of the processors when executing tasks,to shorten the slack time between tasks.

However, the algorithm 2 and algorithm 3 are not exactly the same as that introduced by [7]. We adjust the algorithms a little bit to make them more clear and more efficient than those in [7].

We adjust the estimated finish time of the tasks on the priority list with the following equation. After adjusting the estimated finish time of one task on the priority list, we use algorithm 1 again to calculate the estimated start time and estimated finish time of the rest tasks on the priority list, in case that there is any changing of processor assignment, estimated start time and estimated finish time due to the adjustment. This is our adjustment on the algorithm which is different from that introduced by [7].

$$DDL(n_i, p_j) = EFT(n_i, p_j) * SL_{con}(A)/SL_{actual}(A) \tag{6}$$

After adjusting each task's finish time, it is greatly possible that $DDL(n_i, p_j) > EFT(n_i, p_j)$. So there is a gap between $DDL(n_i, p_j)$ and $EFT(n_i, p_j)$. We can extend the $EFT(n_i, p_j)$ to $DDL(n_i, p_j)$ by adjusting the processor's frequency which can cut down the energy consumption. The energy consumption for task n_i executed on processor p_k with frequency $f_{k,h}$ can be calculated by:

$$E(n_i, p_k, f_{k,h}) \\ = (Pow_{k,ind} + C_{k,ef} * f_{k,h}^{m_k}) * T(n_i, p_k) * f_{k,max}/f_{k,h} \quad (7)$$

$Pow_{k,ind}$ is the static power consumption of p_k which is frequency-independent. $C_{k,ef}$ is the effective switching capacitance of p_k. m_k is the dynamic power exponent of p_k. $f_{k,max}$ and $f_{k,h}$ are the maximum frequency and actual frequency of p_k respectively. $T(n_i, p_k)$ is the execution time of n_i executed on p_k with its maximum frequency.

The total energy consumption of application A is:

$$E(A) = \sum_{i=1}^{N} (E(n_i, p_k, f_{k,h}) \quad (8)$$

By adjusting the frequency of the processor, the estimated finish time of n_i can be calculated by:

$$EFT(n_i, p_j) = EST(n_i, p_j) + T(n_i, p_j) * f_{j,max}/f_{j,h} \quad (9)$$

Algorithm 2 will change scheduling of each task by adjusting the frequency of each processor based on the result of algorithm 1.

```
Data: task priority list $qlist$, $T(n_i,p_k)$, $c_{m,i}$,
      $EST(n_i,p_k)$ and $EFT(n_i,p_k)$ from Algorithm
      1
Result: $n_i$ is assigned to $p_j$, frequency $f_{j,h}$, updated
        $EST(n_i,p_k)$, updated $EFT(n_i,p_k)$, updated
        $SL_{actual}(A)$, total energy consumption $E(A)$
$E(A) \leftarrow 0$
while $qlist \neq \emptyset$ do
    $n_i$ = qlist.pop()
    Compute $DDL(n_i,p_j)$ (6)
    $E(n_i,p_j) \leftarrow \infty$
    for $f_j, h \in F_j$ do
        Compute $EFT(n_i,p_j)$ (9) and $E(n_i,p_j,f_{j,h})$
         (7)
        if $EFT(n_i,p_j) \leq DDL(n_i,p_j)$ AND
         $E(n_i,p_j,f_{j,h}) \leq E(n_i,p_j)$ then
            Assign frequency h to processor j
            $E(n_i,p_j) \leftarrow E(n_i,p_j,f_{j,h})$
            break
        end
    end
    $E(A) \leftarrow E(A) + E(n_i,p_j)$
    With the new $EFT(n_i,p_j)$ use algorithm 1 to
     assign the rest tasks in $qlist$.
end
```

Algorithm 2: The DUPRS algorithm part 1

The complexity of algorithm 2 is $O(N * F * E * P)$, where N is the number of tasks in the application, F is the number of frequencies for a processor.

Algorithm 2 tries to adjust the estimated finish time starting from the entry task to reduce energy consumption, while algorithm 3 is to adjust the estimated finish time starting from the exit task to further reduce the gap between start time and finish time of each tasks and then reduce the energy consumption.

$$DDL2(n_i, p_j) \\ = min\{min_{n_m \in succ(n_i)}(EST(n_m) - c_{m,i}), EST(n_{dn(i)})\} \\ DDL2(n_{exit}, p_j) = EFT(n_{exit}, p_j) \quad (10)$$

$n_{dn(i)}$ is the task intermediate after n_i on the same processor. n_m is the intermediate successor of n_i.

After getting the new estimated finish time, then the estimated start time can be updated with:

$$EST(n_i, p_j) \\ = EFT(n_i, p_j) - T(n_i, p_j) * f_{j,max}/f_{j,h} \quad (11)$$

```
Data: task priority list $qlist$, $T(n_i,p_k)$, $c_{m,i}$,
      $EST(n_i,p_k)$ and $EFT(n_i,p_k)$ from Algorithm
      2
Result: $n_i$ is assigned to $p_j$, frequency $f_{j,h}$, updated
        $EST(n_i,p_k)$, updated $EFT(n_i,p_k)$, updated
        $SL_{actual}(A)$, updated total energy
        consumption $E(A)$
$E(A) \leftarrow 0$
sort $qlist$ in ascending order
while $qlist \neq \emptyset$ do
    $n_i$ = qlist.pop()
    if $n_i == n_{exit}$ then
        continue
    end
    Compute $DDL2(n_i,p_j)$ (10)
    if $EFT(n_i,p_j) \leq DDL2(n_i,p_j)$ then
        $E(n_i,p_j) \leftarrow \infty$
        for $f_j, h \in F_j$ do
            Compute $EFT(n_i,p_j)$ (9) and
             $E(n_i,p_j,f_{j,h})$ (7)
            if $EFT(n_i,p_j) \leq DDL2(n_i,p_j)$ AND
             $E(n_i,p_j,f_{j,h}) \leq E(n_i,p_j)$ then
                Assign frequency h to processor j
                $E(n_i,p_j) \leftarrow E(n_i,p_j,f_{j,h})$
                break
            end
        end
        Compute $EST(n_i,p_j)$ (11)
    end
    $E(A) \leftarrow E(A) + E(n_i,p_j)$
end
```

Algorithm 3: The DUPRS algorithm part 2

The complexity of algorithm 3 is $O(N * F)$.

After algorithm 3, the finalized schedule for each task and the minimized consumed energy are generated.

V. Experiments and Analysis

The purpose of the experiments is to test that Oakestra with energy-efficient scheduling method performs better than the state-of-the-art in energy consumption. We build a geo-distributed and heterogeneous edge computing system with one root, and three clusters at different geographical locations in east area of US. Each cluster consists of 3 worker nodes, which are Raspberry Pi devices (Raspberry Pi4 4GB RAM, Raspberry Pi4 2GB RAM, Raspberry Pi3B+ 1GB RAM).

For the state-of-the-art, we select K3S as the baseline. As mentioned in earlier Section II and in the Oakestra paper, we found that K3s is widely considered to be the de-facto orchestration engine for IoT/edge devices. Therefore, we sought to understand the power and energy consumption of K3s in comparison as well.

K3s, distinguished as a fully compliant Kubernetes distribution, is particularly tailored for Edge Devices, offering key enhancements:

- Single Binary Packaging: K3s is conveniently packaged as a single binary, simplifying deployment.
- Streamlined TLS Handling: The inclusion of a straightforward launcher handles the intricacies of TLS and various options.
- Default Security Measures: K3s is inherently secure, featuring sensible defaults tailored for lightweight environments.
- Reduced External Dependencies: Dependencies have been minimized, necessitating only a modern kernel and cgroup mounts.

To validate the correctness of our hypothesis, we first run various measurements for the edge devices using different methodologies. Each methodology is described below along with accompanying measurements visualizations. Our primary focus is to evaluate the power consumption of the Raspberry Pi devices; therefore, we utilize a USB power meter to measure energy usage. The meter provides us with two different power readings: Voltage and Current. We calculate the energy consumption of the device by calculating Power which is the product of Voltage times Current.

A. Baseline Power Consumption of Idle Hardware

Across all three measurements we can observe that the Voltage remains close to constant. There are minor fluctuations in the measured Current which results in spikes on the Power measurement as well. Across multiple measurements we saw an average of 1.866 Watts for the Raspberry Pi 4B 4GB, 1.793 Watts for the Raspberry Pi 4B 2GB, and 2.216 Watts for the Raspberry Pi 3B+.

B. Power Consumption of Native Program Execution

Before profiling the energy and power consumption of the respective virtualization methods and scheduling algorithms, we wanted to profile the power consumption when running a native executable. Our hypothesis is that virtualization and improved scheduling algorithms yield energy savings; therefore it was important to capture what we see as the ”base case”.

To this end we ran the stress program which is a ”tool to impose load on and stress test a computer system” [27]. We chose this program because it generates workload for us and it allowed us the configure the number of CPUs, memory, I/O and disk stress on our target Raspberry Pi machines.

On average, the power consumption was 5.287 Watts for the Raspberry Pi 4B 4GB, 5.013 Watts for the Raspberry Pi 4B 2GB, and 2.935 Watts for the Raspberry Pi 3B+.

Our investigation into energy consumption extended to K3S with Argo Workflow [28]. When running the stress program in Argo, we saw an average power consumption of 4.551 Watts for the for the Raspberry Pi 4B 4GB, 4.928 Watts for the Raspberry Pi 4B 2GB, and 4.732 Watts for the Raspberry Pi 3B+.

C. Power Consumption of the Original Oakestra System

To determine the power consumption of edge devices when using Oakestra, we used a similar method to our ”base case” scenario. In this case we tested with a Docker container which ran the ‘stress‘ workload generator. We utilized Oakestra to deploy these containers onto the three worker nodes.

The cluster formed to evaluate this power consumption consisted of one root orchestrator on our personal PC, a cluster orchestrator on the same personal PC, and the three raspberry Pi worker nodes. The worker nodes are networked together to the root and cluster orchestrators via Oakestra’s provided NetworkManager component.

The Docker container containing the ‘stress‘ executable was deployed to the cluster using Oakestra’s provided frontend. The results of running the container in the three different worker nodes are provided below.

Fig.3, Fig.4, and Fig.5 detail the power consumption of the 3B+, 4B (2GB), and 4B (4GB) nodes over time. We measured 100 percent CPU utilization in each of the nodes.

Across multiple measurements we saw an average of 4.723 Watts for the Raspberry Pi 4B 4GB, 4.776 Watts for the Raspberry Pi 4B 2GB, and 2.953 Watts for the Raspberry Pi 3B+.

We observe that on average, running the stress application through Oakestra consumed less power on average than running natively for the 4B boards and was about the same for the 3B+ board.

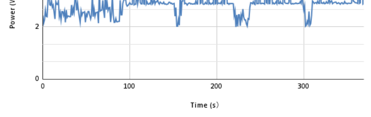

Fig. 3: Power consumption of Raspberry Pi 3B+

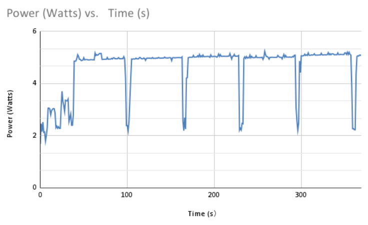

Fig. 4: Power consumption of Raspberry Pi 4B (2 GB)

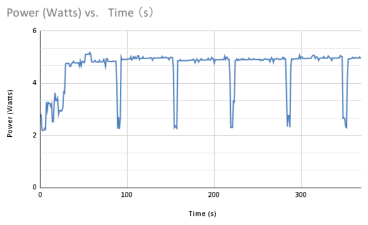

Fig. 5: Power consumption of Raspberry Pi 4B (4GB)

D. Simulation setting

We compare the performance among K3S, Oakestra with its original scheduling method and the energy-efficient scheduling method.

As mentioned in Section III, Oakestra's original scheduling method is LDP which is based on ROM. For the ROM, resource selection strategy should be determined by user. Here we choose the resource selection strategy of picking the first worker which satisfies the requirements in SLA.

We test on three DAG applications, one is the same as the one shown in Fig.2 which has 10 microservices, and the other two are the popular applications in real world: Fast Fourier transform (FFT), Gaussian elimination (GE). The FFT application is with high parallelism while the GE application is with low parallelism. The FFT application has 16 microservices, while the GE application has 14 microservices.

The microservices in the DAG applications are designed following the method introduced by [26]. There are 3 types of microservices, and their complexity are of $O(a*N)$, $O(a*NlogN)$, and $O(N*N)$ separately. Here we set $N = 4096$ and $a = 82$. We picked microservices randomly from these three types to form the DAG applications.

E. Simulation results and analysis

Our experiments are mainly comparing the scheduling length and energy consumption of K3S, Oakestra with its original scheduling and the energy-efficient scheduling method (HEFT + DUPRS part1 + DUPRS part2). We simulate the energy-efficient scheduling method working on Oakestra. The github link to the simulator is [*redacted for double blind review*]. We record the results from different stages, HEFT, DUPRS part1 and DUPRS part2. (The unit for scheduling length is ms, and the unit for energy consumption is mW for Fig.6, 7, and 8.) Furthermore, we also generate the timeline for each task at each phase (HEFT + DUPRS part1 + DUPRS part2). (The unit for the timeline is ms in Fig. 9.)

Fig.6 shows the scheduling length and the energy consumption of DAG application shown in Fig.2. It shows that K3S performs almost the same as Oakestra with its original scheduling. K3S has 9 worker nodes. Oakestra also has 9 worker nodes, but after the coarse scheduling of the root orchestrator, there are actually 3 worker nodes in the scheduled cluster to execute the microservices. Even though there is difference in the number of standby worker nodes for executing the microservices, Oakestra has the similar scheduling length as that of K3S. The difference in scheduling length is just 1%. This is because the communication cost between worker nodes in different geographical locations has a negative impact on the performance even though there are more options for selection. This is an advantage of Oakestra scheduling especially for communication-intensive tasks. On the other hand, Oakestra consumes 2% more energy than K3S. However, we can see that HEFT has shorter scheduling length than both K3S and Oakestra with its original scheduling, even though in K3S has 9 worker nodes while Oakestra cluster only has 3 workers. In addition to the reason that the communication cost between worker nodes in different geographical locations brings negative impact to K3S, the HEFT algorithm uses the priority list of the tasks for scheduling, which optimizes the consequence to process each microservice. On the other side, the energy consumption of HEFT is higher than that of K3S. In the phases of DUPRS part1 and part2, we use 1400 as the threshold for scheduling length. In these two steps, we can see that there is a great reduction in energy consumption. In the result of DUPRS part2, the scheduling length is increased by 2% compared with Oakestra with its original scheduling, but the energy consumption is reduced by 43%, because the DUPRS takes a good use of the slack time between microservices to reduce the energy consumption by adjusting the frequency of processors when executing tasks. In addition, the performance of DUPRS part2 is much better than K3S. Even though the scheduling length is increased by 3% which is too small and can be ignored, the energy consumption is reduced by 45%.

Fig.9 shows that how DUPRS part1 and DUPRS part2 adjust the execution time of each task by adjusting the frequency of each processor. DUPRS part1 extends the execution time of each task to meet the scheduling length threshold. DUPRS part2 it shortens the gap between the tasks. As we see in the phase of HEFT and DUPRS part1, there is a gap between task 5 and task 9, but in the phase of DUPRS part2, there is no such a gap. In addition, the execution time of task 9 is extended a lot. It is also a shortening of gap because task 9 is the predecessor of task 10. In the phase of HEFT and DUPRS part1, there is a big gap between task 9 and task 10. However, in the phase of DUPRS part2, the gap is greatly shortened. Because the execution time of each task is extended, the total energy consumption is reduced.

Fig.7 and Fig.8 show the same results as that of Fig.6. For these three types of DAG applications, the energy-efficient

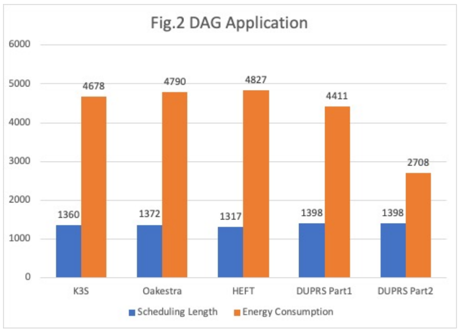

Fig. 6: Scheduling Length and Energy Consumption for DAG application in Fig. 2.

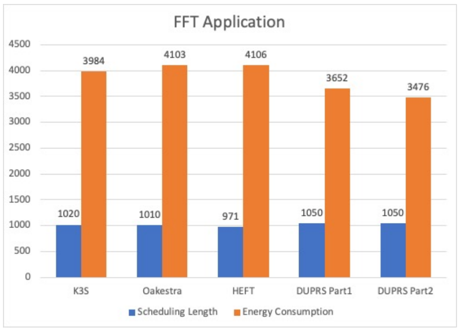

Fig. 7: Scheduling Length and Energy Consumption for FFT Application

scheduling method increases scheduling length by 2%-9% compared with Oakestra with its original scheduling, while it reduces the energy consumption by 15%-43%. In addition, for these three types of DAG applications, the energy-efficient scheduling method increases scheduling length by 3%-7% compared with K3S, but the energy consumption is reduced by 16%-45%. So for these DAG applications, the energy-efficient scheduling outperforms both K3S and Oakestra with its original scheduling in energy consumption with an attractive tradeoff in scheduling length.

VI. Future Work and Conclusion

A. Future Work

It would be beneficial to explore the adaptability of the proposed energy-efficient scheduling method through a broader range of edge computing scenarios and applications. Investigating the scalability of the proposed energy-efficient scheduling method with Oakestra to handle larger and more diverse edge device environments is crucial. Furthermore, a comprehensive experimental examination of the energy-efficient scheduling method's performance and robustness under varying workloads and network conditions would help towards its practical applicability. Efforts in optimizing the orchestration framework for emerging technologies and evolving hardware configurations will be integral for sustained advancements in the field of energy-efficient distributed edge computing.

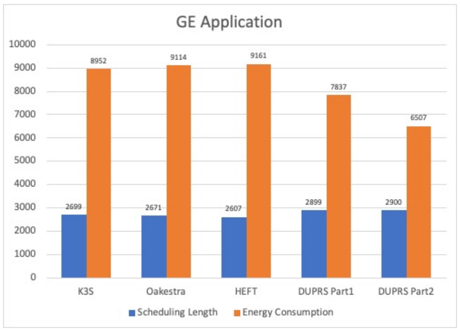

Fig. 8: Scheduling Length and Energy Consumption for GE Application

B. Conclusion

In summary, our paper contributes to the practical application of container orchestration in edge computing. We offer valuable insights into the selection of container orchestration solutions for DAG applications and set a benchmark for future research in this burgeoning field. As we move towards an era where edge computing plays an increasingly pivotal role in real-time analytics and systems, IoT, and autonomous systems, we offer a path forward for more efficient, sustainable, and high-performing edge computing solutions that have the potential to reshape industries and improve the lives of people worldwide.

Our detailed analyses of DAG applications demonstrate that the proposed energy-efficient scheduling method surpasses the performance of both K3S and Oakestra with its original scheduling. The presented simulation results reveal substantial improvements, 15%-43% decrease in energy consumption for the energy-efficient scheduling method compared to its original counterpart. Notably, the energy-efficient scheduling method achieves remarkable energy savings of 16%-45% relative to K3S, underscoring its efficacy in addressing the challenges posed by power-limited edge devices.

Our research demonstrates the superior performance of Oakestra with the energy-efficient scheduling method and also signifies a crucial step forward in advancing energy-efficient orchestration frameworks tailored for geo-distributed edge computing environments. By addressing the critical issue of energy consumption in edge devices, our work lays the foundation for more sustainable and impactful edge computing solutions.

References

[1] Sahni, Y., Cao, J., Yang, L., & Wang, S. (2022). Distributed resource scheduling in edge computing: Problems, solutions, and opportunities. Computer Networks, 219, 109430.

[2] T. Holmes, C. McLarty, Y. Shi, P. Bobbie and K. Suo, "Energy Efficiency on Edge Computing: Challenges and Vision," 2022 IEEE International Performance, Computing, and Communications Conference (IPCCC), Austin, TX, USA, 2022, pp. 1-6, doi: 10.1109/IPCCC55026.2022.9894303.

[3] C. Wang, X. Yu, L. Xu and W. Wang, "Energy-Efficient Task Scheduling Based on Traffic Mapping in Heterogeneous Mobile-Edge Computing: A Green IoT Perspective," in IEEE Transactions on Green Communi-

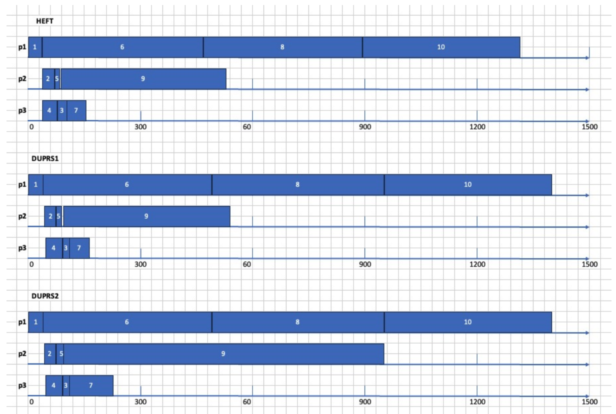

Fig. 9: Timelines for Each Task at Each Phase for DAG application in Fig. 2.

cations and Networking, vol. 7, no. 2, pp. 972-982, June 2023, doi: 10.1109/TGCN.2022.3186314.

[4] Giovanni Bartolomeo, Simon Bäurle, Nitinder Mohan, and Jörg Ott. 2022. Oakestra: an orchestration framework for edge computing. In Proceedings of the SIGCOMM ’22 Poster and Demo Sessions (SIGCOMM ’22). Association for Computing Machinery, New York, NY, USA, 34–36. https://doi.org/10.1145/3546037.3546056

[5] Alves, M.P., Delicato, F.C., Santos, I.L. et al. LW-CoEdge: a lightweight virtualization model and collaboration process for edge computing. World Wide Web 23, 1127–1175 (2020). https://doi.org/10.1007/s11280-019-00722-9

[6] Kim, S.-H.; Kim, T, ”Local Scheduling in KubeEdge-Based Edge Computing Environment”, Sensors 2023, 23, 1522. https://doi.org/10.3390/s23031522

[7] Jing Liu, Pei Yang, Cen Chen, ”Intelligent energy-efficient scheduling with ant colony techniques for heterogeneous edge computing”, Journal of Parallel and Distributed Computing, Volume 172, 2023, Pages 84-96, ISSN 0743-7315

[8] G. Xie, J. Jiang, Y. Liu, R. Li and K. Li, ”Minimizing Energy Consumption of Real-Time Parallel Applications Using Downward and Upward Approaches on Heterogeneous Systems,” in IEEE Transactions on Industrial Informatics, vol. 13, no. 3, pp. 1068-1078, June 2017, doi: 10.1109/TII.2017.2676183.

[9] S. K. Panda, M. Lin and T. Zhou, ”Energy-Efficient Computation Offloading With DVFS Using Deep Reinforcement Learning for Time-Critical IoT Applications in Edge Computing,” in IEEE Internet of Things Journal, vol. 10, no. 8, pp. 6611-6621, 15 April15, 2023, doi: 10.1109/JIOT.2022.3153399.

[10] B. Hu, Y. Shi and Z. Cao, ”Adaptive Energy-Minimized Scheduling of Real-Time Applications in Vehicular Edge Computing,” in IEEE Transactions on Industrial Informatics, vol. 19, no. 5, pp. 6895-6906, May 2023, doi: 10.1109/TII.2022.3207754.

[11] M. S. Aslanpour, A. N. Toosi, M. A. Cheema and R. Gaire, ”Energy-Aware Resource Scheduling for Serverless Edge Computing,” 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 2022, pp. 190-199, doi: 10.1109/CCGrid54584.2022.00028.

[12] Y. Li and S. Wang, ”An Energy-Aware Edge Server Placement Algorithm in Mobile Edge Computing,” 2018 IEEE International Conference on Edge Computing (EDGE), San Francisco, CA, USA, 2018, pp. 66-73, doi: 10.1109/EDGE.2018.00016.

[13] L. Cui et al., ”Joint Optimization of Energy Consumption and Latency in Mobile Edge Computing for Internet of Things,” in IEEE Internet of Things Journal, vol. 6, no. 3, pp. 4791-4803, June 2019, doi: 10.1109/JIOT.2018.2869226.

[14] M. Daraghmeh, I. Al Ridhawi, M. Aloqaily, Y. Jararweh and A. Agarwal, ”A Power Management Approach to Reduce Energy Consumption for Edge Computing Servers,” 2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC), Rome, Italy, 2019, pp. 259-264, doi: 10.1109/FMEC.2019.8795328.

[15] Jozef Mocnej, Martin Miškuf, Peter Papcun, Iveta Zolotová, ”Impact of Edge Computing Paradigm on Energy Consumption in IoT”, IFAC-PapersOnLine, Volume 51, Issue 6, 2018, Pages 162-167, ISSN 2405-8963

[16] E. Ahvar, A. -C. Orgerie and A. Lebre, ”Estimating Energy Consumption of Cloud, Fog, and Edge Computing Infrastructures,” in IEEE Transactions on Sustainable Computing, vol. 7, no. 2, pp. 277-288, 1 April-June 2022, doi: 10.1109/TSUSC.2019.2905900.

[17] H. Topcuoglu, S. Hariri and Min-You Wu, ”Performance-effective and low-complexity task scheduling for heterogeneous computing,” in IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pp. 260-274, March 2002, doi: 10.1109/71.993206.

[18] S. Azizi, M. Othman and H. Khamfroush, ”DECO: A Deadline-Aware and Energy-Efficient Algorithm for Task Offloading in Mobile Edge Computing,” in IEEE Systems Journal, vol. 17, no. 1, pp. 952-963, March 2023, doi: 10.1109/JSYST.2022.3185011.

[19] X. Lin, Y. Wang, Q. Xie and M. Pedram, ”Energy and Performance-Aware Task Scheduling in a Mobile Cloud Computing Environment,” 2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, 2014, pp. 192-199, doi: 10.1109/CLOUD.2014.35.

[20] H. Ma, P. Huang, Z. Zhou, X. Zhang and X. Chen, ”GreenEdge: Joint Green Energy Scheduling and Dynamic Task Offloading in Multi-Tier Edge Computing Systems,” in IEEE Transactions on Vehicular Technology, vol. 71, no. 4, pp. 4322-4335, April 2022, doi: 10.1109/TVT.2022.3147027.

[21] Michael Pendo John Mahenge, Chunlin Li, Camilius A. Sanga, ”Energy-efficient task offloading strategy in mobile edge computing for resource-intensive mobile applications”, Digital Communications and Networks, Volume 8, Issue 6, 2022, Pages 1048-1058, ISSN 2352-8648

[22] T. Gizinski and X. Cao, ”Design, Implementation and Performance of an Edge Computing Prototype Using Raspberry Pis,” 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 2022, pp. 0592-0601, doi: 10.1109/CCWC54503.2022.9720848.

[23] S. Wolfgang and X. Cao, ”Raspberry Pi Based Computing Prototypes: Design, Implementation and Performance Analysis,” 2023 IEEE International Conference on Electro Information Technology (eIT), Romeoville, IL, USA, 2023, pp. 059-066, doi: 10.1109/eIT57321.2023.10187384.

[24] Oakestra Github Repository: https://github.com/oakestra

[25] H. Topcuoglu, S. Hariri and Min-You Wu, ”Performance-effective and low-complexity task scheduling for heterogeneous computing,” in IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pp. 260-274, March 2002, doi: 10.1109/71.993206.

[26] P. -F. Dutot, T. N’Takpé, F. Suter and H. Casanova, ”Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms,” in IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 7, pp. 940-952, July 2009, doi: 10.1109/TPDS.2009.11.

[27] Stress: “https://manpages.ubuntu.com/manpages/lunar/en/man1/stress.1.html”.

[28] Argo Link: ”https://argoproj.github.io/argo-workflows/”