Events as a Basis for Workflow Scheduling

David Marchant *Datalogisk Institut (DIKU), Copenhagen University*
Copenhagen, Denmark
d.marchant@ed-alumni.net, 0000-0003-4262-7138

Abstract—This paper evaluates using event-based scheduling as a basis for dynamic workflow management. To do this, the `WorkflowRunner` is introduced as a tool for conducting event driven scheduling in a robust manner. It is evaluated in comparison to Slurm and the `WorkflowRunner` is found to schedule analysis roughly 2.5x quicker than Slurm in most cases. An example workflow is also presented, demonstrating how this style of scheduling allows for complete modification of the workflow structure at runtime, something very difficult to achieve in traditional workflow management systems. These developments are expected to be of particular use in distributed analysis systems, and in heterogeneous systems such as those looking to accommodate human-in-the-loop interactions.

Keywords—Dynamic, Heterogeneous, MEOW, Workflows.

I. Introduction

Modern scientific research frequently involves the processing of large amounts of experiment data, in long running jobs, on dedicated hardware. This analysis often comprises of several different steps and is can be managed by Scientific Workflow Management Systems (SWMSs). These typically use Directed Acyclic Graphs (DAGs) as a basis for composing a workflow, with individual steps linked together in a linear progression. A DAG allows SWMSs to easily identify different jobs and their dependencies within a workflow, as well as making workflow composition straightforward for users. However, scientific workflows are exploratory in nature [10], and have an inherent need to be dynamic [24]. This need is not easily met through the use of a static DAG, and so alternatives are needed.

One alternative to the static DAG paradigm is the use of dynamic schedulers, where jobs are scheduled individually without a full workflow necessarily being understood at the start. Managing Event Oriented Workflows (MEOW) [23] is an example of this. It uses event monitors to respond to file events by scheduling individual jobs as part of a continuous, live system. This can potentially distributes control of the scheduling across multiple nodes, and enables dynamic processing that is not possible within a static, sequential system.

A. Existing SWMS Offerings

Unsurprisingly, a large number of tools for constructing and managing scientific workflows already exist. Many of these are dedicated tools such as Apache Airflow [18], Kepler [1], Pegasus [9], Taverna [29], Dask [34], DagOn* [26], Askalon [15], and DVega [36]. All of these systems use a data-flow model, where a pipeline of processing is constructed and data is passed from process to process. Each process will have defined inputs and outputs, and should always perform some modification on the input data. This model is maintained through the use of one or more DAGs. A DAG is a linked graph in which nodes are directionally connected such that a loop is never formed. Thus, a DAG is an easy analogue for a workflow, with a defined start and end, and with the dependencies between the different steps trivial to identify. The DAGs themselves are constructed by the user either through a web interface, or programatically. More than one DAG may be provided as individual steps within the workflow may be smaller workflows themselves.

Typically, the previously mentioned systems do not carry out processing themselves, but schedule jobs to be executed using tools such as Kubernetes [21], Slurm [42], Globus [17], corc [27], WLM [40], Torque [37], UNICORE [4], parsl [2], cwltool [7], MapReduce [8], or OGE [30]. These can each also be used on their own, though often have fewer usability aids such as a GUI. They also typically offer a lower-level control of job scheduling and frequently will have fewer extra features such as robust error handling. Several of these systems can be used in conjunction to create sub-workflows, though few have specific accommodation for sub-workflows not written in the same system as themselves. A note-able exception to this is the ambiguously named Hybrid Workflows system built on COMPS [3], presented in [41] and [33]. This presents two types of workflow, in-situ and task-based. In-situ workflows are run within a single resource. Task-based workflows however are large, task parallel batches of processing that may take place over all manner of remote resources, or be processed locally. Hybrid Workflows uses Decaf [12] to run in-situ workflows on performance systems. These in-situ workflows are managed by PyCOMPS [35], which runs each Decaf workflow as a step in a larger task-based workflow. This allows for a large chain of analysis, with individual steps tailored to their specific hardware needs, and allows users to exploit the benefits of both types of workflow.

One final workflow system that is worth considering is WED-flows [16]. WED-flows is interesting as it is an event-driven workflow system. In WED-flows, data is processed according to user defined trigger conditions. From one of these triggers a series of processing activities are started, in a control-flow fashion. The descriptions of WED-flows [16] are unclear on whether a DAG is specifically used or not in the creation and scheduling of processing tasks. It would certainly be a task appropriate for a DAG, and there is no more dynamic scheduling within those processing tasks than in any other SWMS. These sub-workflows themselves are not event-driven, and it is only the scheduling of the entire workflow itself that

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604.

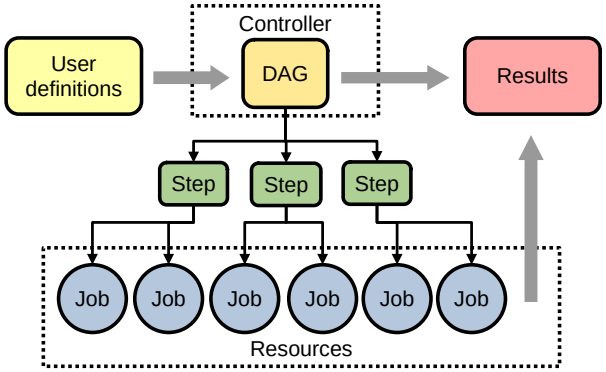

Fig. 1. A top-down workflow structure. Not shown is any data transfer or dependency between jobs.

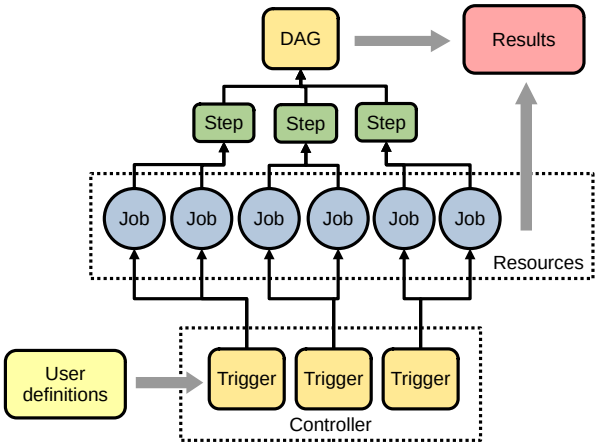

Fig. 2. A bottom-up workflow structure. Not shown is any data transfer or dependency between jobs.

is undertaken in response to an event.

Formally, all of these systems could be characterised as a top-down approach as a central workflow controller is determining the entire structure ahead of time and dictating it to individual jobs, as is shown in Figure 1. Here we can see that a user submits their workflow definitions to the system, either via a DAG, or something that from which a DAG can be derived. In either case the workflow controller will use this DAG to derive the steps. These steps in turn become the basis for identifying jobs, which are scheduled on resources. No data transfer or dependency between jobs has been shown in Figure 1, but at some point the final job will complete and any output will become part of the workflow results. The DAG will also form part of the results, independent of any job processing, as it is a very informative and crucial part of the workflow construction.

This top-down system of scheduling does not properly accommodate the need for scientific workflows to by dynamic [24]. This is as scientific workflows are inherently exploratory in nature, and so analysis should be capable of being changed at runtime [10]. Workflows should be capable of looping, branching, or otherwise being adapted at runtime in response to new knowledge or errors in execution. Many of the previously listed systems have some accommodation for some of these systems, but they are often imperfect patches onto an inherently static design.

B. Dynamic, Event-based Scheduling

An alternative design would be a bottom-up approach. Rather than a single controller identifying, scheduling and assigning all workflow jobs, individual jobs will be scheduled in isolation. By this it is meant that the system does not actively construct an entire workflow ahead of time, merely that it identifies individual jobs, as is shown in Figure 2. In this Figure we can see that as before, a user provides some definitions. These are not reduced to a DAG but are instead used as the basis for individual job scheduling. Once jobs have completed, they may be linked together into steps. These steps, along with any output from the jobs can then form the results of the *emergent* workflow. The word emergent has been highlighted as it is central to this new approach, and is fundamental to understanding it. To re-iterate, in a bottom-up workflow system, a user does not actually construct a workflow directly, they merely create the conditions such that individual jobs can be scheduled.

One system that adopts a bottom-up design is MEOW [23]. Within MEOW users schedule jobs by defining blocks of analysis code, referred to as a *Recipe*, along with the the conditions under which said processing is scheduled, referred to as a *Pattern*. Any combination of valid *Patterns* and *Recipes* are called *Rules.* The conditions that *Rules* respond to are events. For example, a *Recipe* might be some segmentation algorithm while a *Pattern* might identify raw image data being created at a particular file system location. If both the *Pattern* and *Recipe* are defined then a *Rule* will be created that any raw data created at that location will result in the automatic scheduling of the segmentation analysis on said data. If the raw data is ever updated then the analysis is also automatically re-run. The analysis should produce some output, which may trigger further *Rules*, and so form a chain of processing. This will form the design shown in Figure 4.

For an example of where this may be used consider the workflow structure shown in Figure 3. This shows a complete scientific analysis made up of six distinct systems, with the analysis within it scheduled in response to file events, such as a data file being written as some processing output. Currently, workflows like this exist at a variety of levels in scientific processing. For instance, this could represent many of the experiments carried out at large scientific experiment institutions such as EuXFEL [13] or MAX IV [25], In these typically researchers gather large amounts of data from instruments which must be initially cleaned up before being passed into some local storage. This local storage is limited and so data will need to be moved to longer term storage from where researchers will schedule varying forms of analysis using a variety of processing platforms, usually not local to the data storage. This style of workflow is not limited to large institutions, with smaller experiments using a similar structure of having to gather data then moving it between many specialised storage and compute systems throughout the lifetime of workflow [38].

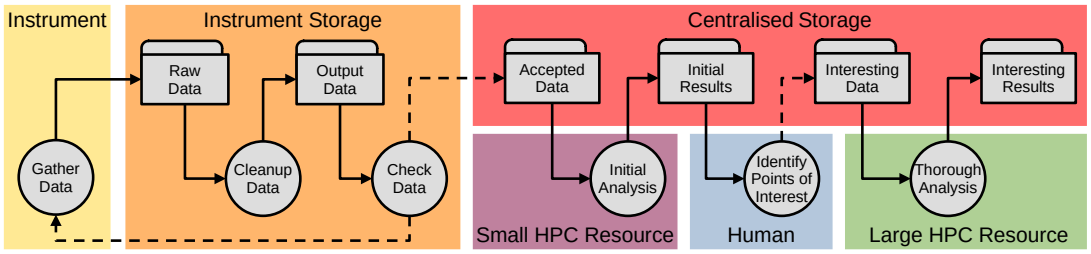

Fig. 3. A heterogeneous analysis. Data storage is shown as folders while processing, human interaction or the running of hardware shown as circles. In this example, scheduling would be any line leading to a circle. Note that solid lines show where all data inputs will produce output, while dotted lines show where only some inputs will produce outputs.

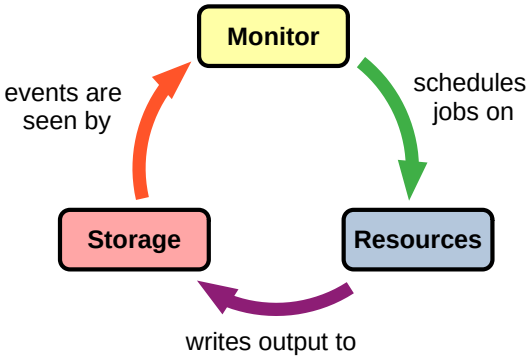

Fig. 4. MEOW's event-driven design structure.

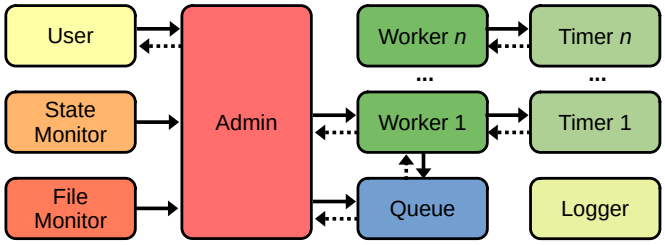

Fig. 5. Process structure of the `WorkflowRunner`, showing individual processes and their interactions. Connections to the logger have not been omitted for brevity. Secondary connections used only for replies are shown in dotted lines. Zero to n workers are created based on user input.

MEOW is well suited to this system for three core reasons. Firstly, in such a distributed system there is no obvious point of central control from which a more traditional SWMS can operate to accommodate the entire workflow from end to end. For example, while much analysis and storage can be managed remotely from a users personal workstation, the operation of scientific instruments often cannot be. Similarly, many dedicated analysis systems require access from on site, or from within specific networks.

Secondly, as there is more demand for heterogeneous systems or human-in-the-loop interactions, it will become more difficult to manage such a range of resources from a single script or tool. Whilst tools such as CUDA exist to hand off processing from a CPU to a GPU, such tools do not always exist for every piece of dedicated hardware or instrument. A lower level, generic approach such as reacting to data being produced simplifies what would otherwise be complex communication between different hardware systems.

Third and lastly, as the platforms for analysis become more disjointed, there is an expected to be an increased likelihood of errors throughout the workflow, especially in light of their exploratory nature. Moving data across networks is expensive in both time and data-usage and so repeated processing of already correct data should be avoided. By completing each job in isolation, MEOW can easily maintain completed analysis without one single error discarding an entire run of processing.

II. A Workflow Runner

MEOW was designed initially to function as part of a grid management solution, the Minimum intrusion Grid (MiG) [5]. The MiG is a large grid solution, and so obviously not suited to be deployed onto a wide variety of resources. Therefore, if we are to use the MEOW system outlined within it on varied hardware, then a new, smaller tool that only implements the event driven scheduling would be needed. To do this, a construct was created within the Python package `mig_meow` [22] for running MEOW systems locally on any machine capable of running Python 3. This tool was given the name `WorkflowRunner`. It is intended as a proof of concept of how event driven scheduling implementations can be used on individual resources, but also as a robust and performant implementation in its own right. Therefore it was designed with a parallel internal structure. To ensure freedom from deadlock and livelock a design utilising the principles of Communicating Sequential Processes (CSP) [19] was adopted as shown in Figure 5. As no circular loop of processing interactions exist we can conclude in accordance with CSP that it is impossible for this system to deadlock or livelock.

The USER process: The *User* process is the base process in which the constructor for the `WorkflowRunner` is called, and from which a `WorkflowRunner` object is returned. The `WorkflowRunner` object is then used as the entry point for any user interaction, with each sending an appropriate message to the `Admin` process. A response is always expected from the `Admin`. An exhaustive list of all provided functions will not be provided here, though they include all necessary functions for the creation, updating and deleting of *Patterns* and *Recipes*, and for monitoring the continued status of the `WorkflowRunner`

The STATE MONITOR process: Both the *State Monitor* and *File Monitor* inherit from the

`PatternMatchingEventHandler`, part of the `watchdog` API [39]. This catches file events. All defined *Patterns* and *Recipes* are saved to disk where they can be modified by a user. Any changes are picked up by the *State Monitor* and are reported to the `Admin` process. No response is ever expected from the `Admin`, so the *State Monitor* process should never be blocked, and is therefore always able to process new events.

The FILE MONITOR process: The *File Monitor* is identical to the *State Monitor* process, though rather than the internal directory for *Patterns* and *Recipes*, it monitors the wider file system, .

The ADMIN process: By far the most complex process is the *Admin*. It maintains the in-memory state of the runner, in which all currently registered *Patterns* and *Recipes* are stored and appropriate *Rules* are created. Updates to this state are provided by the *State Monitor* process.The *Admin* process will also receive input from the *File Monitor*. These events will be compared against the current *Rules*. If the event path matches a *Rule*, then the *Admin* will create a new job, and send its ID to the *Queue*.

As well as this core functionality, the *Admin* deals with requests from the *User* process, such as the previously mentioned adding or modifying *Patterns* and *Recipes*. Some requests, such as to query the current queue composition require further messages to be sent to the *Queue* before a response can be generated, but a response is inevitable and provided as soon as possible. The *Admin* process utilises a `wait` statement to hang until receiving input from either the *State monitor*, *User*, or *File Monitor* processes. These three inputs are prioritised in the order given, so that if multiple are available at the same time, only the first is read and processed.

Messages from the *State Monitor* are always of the highest priority as a fresh state will always be needed by the Admin. Changes in the state file system will also be finite in nature. Secondly, are messages from the *User* process, which can be replied to on a human time-frame. This means that they do not need to be responded to within nanoseconds and so can wait behind any *State Monitor* updates. Lastly, this leaves the *File Monitor*. This may produce a theoretically infinite number of messages as there is no limit on the number of files created or updated by jobs. Despite this being an unlikely use case, it is nevertheless a possibility and should be accounted for, therefore it must be the lowest priority as anything behind it could be eternally starved in this scenario.

The WORKER process: Jobs are executed within the *Worker* processes. The amount of these to be spawned is determined by the user, and at least one is needed if the workflow runner is to process jobs. A *Worker* will request a job from the *Queue* process. If a job is available, the ID will be sent to the *Worker*, and it will be executed. The job itself is processed by first parameterizing the input notebook using the python module `notebook_parameterizer` [28]. This is then run using `papermill` [31] in the same manner as is done on the MiG. Once execution has been completed, the job files are copied into a separate job output directory where they can be individually inspected. Jobs may produce output directly into the data directory, monitored by the *File Monitor*, in the same manner as can be done within the MiG. If no job was available in the *Queue*, the *Worker* sends a notification to its *Timer* process to start sleeping. If a job completes, or the *Worker* is notified by the *Timer*, it will poll the *Queue* for another job. This polling of the *Queue* will loop until manually stopped.

	Laptop	Threadripper
Cores	4	16
Hyperthreads	8	32
Processor	i7-8550U	Threadripper 1950X
Clockspeed	1.8 GHz	3.4 GHz
RAM	8GB	112GB
Disk	SSD	SSD

TABLE I
RESOURCES USED THROUGHOUT THIS PAPER.

The TIMER process: To prevent spamming the *Queue* process with requests for new jobs, each *Worker* has its own *Timer* process. This process will wait for a start signal from their *Worker* and then sleep. Once the sleep is over, it will send a signal to the *Worker*. By having the timer in a separate process rather than internal to the *Worker*, the *Worker* is still free to receive messages from the *Admin*, which would not be the case if it itself were sleeping.

The QUEUE process: The *Queue* process acts as a buffer for all jobs that have not yet been processed by a worker. It accepts messages either from the *Admin* or any of the *Worker* processes. From the *Admin*, the *Queue* will either receive the identity of a new job to be added to the queue, or a request for the current composition of the queue. Alternatively, any of the *Workers* may request the identity of a new job to execute. In any case, a response is always immediately generated and sent. It was necessary to separate the queue into its own process, rather than having it stored within the *Admin*, as otherwise there would be a risk of deadlock between *Workers* and the *Admin*.

The LOGGER process: The *Logger* process is just used to send debug messages to either the console or a log file.

III. INVESTIGATING FEASIBILITY

In order to recommend MEOW as a system for scheduling workflow jobs, we should demonstrate that event identification does not add significant overheads to the scheduling process, and that such a system is robust even at scale.

A. Testing `watchdog` Event Identification

As part of demonstrating the robustness of MEOW, we should test that all file events are caught by `watchdog`. Such a test was achieved by starting a monitor process using `watchdog` that would count every file event. Several writer threads would also be started that would concurrently create as many events as possible. Through testing it was determined that on the machine being used for the test, four writers each creating empty files was the fastest way of generating events. The test code is available at [14], with the results of four tests in Table II.

Events Made	Events Seen	Duration (s)	Events (per s)
1,000	1,000	0.35	2857.28
10,000	10,000	3.00	3328.19
100,000	100,000	31.69	3155.38
1,000,000	1,000,000	352.94	2833.31

TABLE II
RESULTS OF THE `WATCHDOG` TEST. ALL RESULTS ARE A MEAN OF 20 RUNS AND ROUNDED TO 2 DECIMAL PLACES. RUN ON THE LAPTOP RESOURCE OUTLINED IN TABLE I.

These results demonstrate that event over long periods of time, `watchdog` does not get swamped and is capable of identifying all file system events. This shows that it is capable of scaling into large systems such as the MiG that may produce large numbers of data files in a short length of time.

B. Measuring MEOW Overheads

To investigate the overheads involved in a MEOW system, we first need a baseline to measure against. Here we will use Slurm. Although not a dedicated SWMS, it is a tool for the mass scheduling of jobs on distributed clusters of resources and so has a similar use case to MEOW. Each of the following tests were run in dedicated Docker [11] containers which can all be found at [14]. All tests were repeated 10 times to get a mean result, and each test was run from 10 to 500 jobs. These tests were not run evenly however, with the amount of jobs increasing at an accelerated rate as the jobs increase. This is as it was expected that the finer trends would be apparent with low amounts of jobs, while the overall relationship should be obvious with only a few larger tests.

Each test was run on both resources outlined in Table I. The laptop is a small machine, representative of the lower powered machines most researchers have as a personal workstation. Meanwhile the threadripper is a custom built machine designed for massive processing of scientific problems. As many of the tests are not explicitly parallelised, there will not be a massive performance difference between the two, but the threadripper will be less prone to being swamped by new threads or context switching, and as a desktop it will have much better cooling capabilities than a laptop. Due to space requirements, only the most pertinent results will be shown here, but full results can be seen at [14].

C. Overheads in Slurm

Slurm has two basic methods of scheduling jobs, `srun` and `sbatch`. Both will schedule one or more jobs, though `srun` is a blocking operation, where a user must wait for all jobs to schedule and execute before their script or terminal can progress. In contrast, `sbatch` schedules jobs in the background and so is a non-blocking operation. This is more akin to how the `WorkflowRunner` and most SWMS work and so `sbatch` will be used to measure against in most future tests.

To get a baseline for Slurm's performance, both `srun` and `sbatch` were used to schedule large numbers of jobs at once and how long it took the scheduling to complete was timed. Note that this does not include the execution time. As MEOW

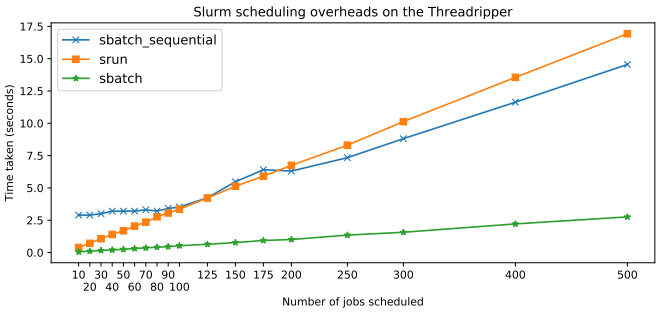

Fig. 6. Slurm scheduling durations on the Threadripper.

can also be used for continuous analysis we should also investigate the overhead of an ongoing system. Another test was therefore developed where jobs were scheduled individually, with each job scheduling a new job. This forms a chain of processing akin to how MEOW is expected to be used. Note that this second test includes execution times as each job needs to be executed to schedule the next. The results of all three tests on the Threadripper are shown in Figure 6. For each test the scheduling time, execution time, and a combination of both are shown. In the case of the `srun` and `sbatch` tests, the scheduling time is the pertinent part to look at and has been highlighted. In the case of the continuous scheduling test it is the combined time, which has also been highlighted. This is as the scheduling is in fact only the timing for scheduling the initial job and so shows an inaccurately fast time, while the combined time is the actual time for all jobs to be scheduled.

As Slurm is a relatively bare-bones system there is not much scope for overhead. This is reflected in the demonstrated very quick scheduling by `sbatch`. As the test was run with only a single Slurm worker, `srun` must wait for jobs to complete before scheduling the next, leading to the much larger overhead. Running `sbatch` sequentially runs is effectively doing the same thing, as each jobs needs to execute to schedule the next, but the use of `sbatch` means that a small amount of concurrency can happen between the scheduler and the processor, hence the slightly decreased overhead as the number of jobs increases. For low job numbers, the overhead in repeatedly starting new threads will slow down the sequential test compared to `srun`. Significantly, in all three tests the scheduling time per job is increasing linearly, meaning that Slurm will scale very well with larger job submissions.

D. Overheads in the `WorkflowRunner`

Within the `WorkflowRunner`, overheads will consist of the sum of the following components: Event identification in `watchdog`, *Rule* lookup, creating one or more new jobs, as well as any associated orchestration overhead. To help identify how much each of these components contribute to the total overhead, as well as to identify additional overheads, five different experiments were created. Unless otherwise noted the *Recipe* in each is some inconsequential execution. The five experiments would are as follows:

- **Single Pattern, Multiple Files (SPMF)**. In this experiment a single *Pattern* would be created that would

trigger on any file event within a directory. *N* files would then be created within the directory, causing the parallel scheduling of *N* jobs. This should allow us to identify the aggregate overheads of MEOW scheduling. This test can be directly compared against the Slurm `sbatch` test.

- **Single Pattern, Single Files, Parallel Scheduling (SPSFP)**. In this experiment a single *Pattern* would be created that would trigger on any file event within a directory. This *Pattern* would include an *N* wide parameter sweep over some variable. A single file would then be created within the directory, causing the parallel scheduling of *N* jobs. By comparing this to SPMF we should be able to identify the overhead caused by `watchdog` and event identification, by minimising it within this experiment. This test can be directly compared against the Slurm `sbatch` test.
- **Single Pattern, Single Files, Sequential Scheduling (SPSFS)**. In this experiment a single *Pattern* would be created that would trigger on any file event within a directory. The *Recipe* used by this *Pattern* would create another file in the same directory, so triggering the *Pattern* again. A variable will also be included so that a file is only created by the first *N*-1 jobs. This will result in sequential scheduling of *N* jobs. This test should illustrate the overhead in continuous, looping scheduling, the most anticipated use-case for a MEOW system. This test can be directly compared against the Slurm sequential `sbatch` test.
- **Multiple Patterns, Single File (MPSF)**. In this experiment *N Patterns* would be created that would trigger on any file event within a directory. A single file would then be created within the directory, causing the parallel scheduling of *N* jobs. When compared against the SPMF experiment, this can isolate the overhead in *Rule* lookup. This test can be directly compared against the Slurm `sbatch` test.
- **Multiple Patterns, Multiple Files (MPMF)**. In this experiment *N Patterns* would be created that would each trigger on a different specific file within a directory. *N* files would then be created within the directory, each matching to a single *Pattern* causing the parallel scheduling of *N* jobs. By comparing this to the other tests we should be able to identify the expected general overhead in a live system, where many differing events may happen at once, as well as the overhead for *Pattern* lookup in a larger memory construct compared to the SPMF test. This test can be directly compared against the Slurm `sbatch` test.

The results for the tests run on the Threadripper are shown in Figure 7 on a logarithmic scale. In the SPMF, MPSF and SPSFP tests the `WorkflowRunner` actually comes out ahead of the Slurm tests, with a speedup of roughly 2.5, though this varies by experiment and the amount of jobs. In each case the per-job scheduling time is reasonably constant demonstrating good scalability in these situations, as can be seen in Figure 8. The MPMF does not scale as well however, with it being slower than the Slurm past roughly 100 jobs

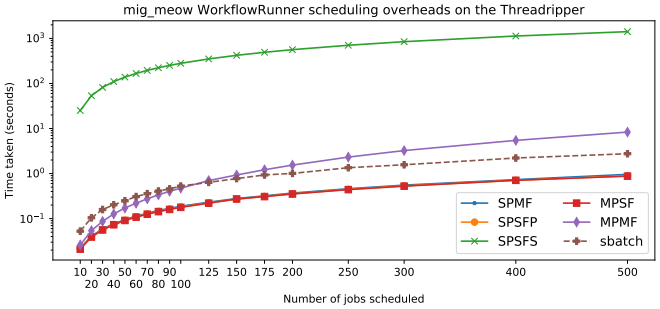

Fig. 7. Logarithmic WorkflowRunner scheduling durations on the Threadripper. The Slurm *sbatch* has also been shown as a dashed line for comparison.

at once. This is down to two parts. Firstly, each file event is identified and processed separately, and secondly it takes time for the `WorkflowRunner` to navigate the stored *Rules*. These overheads each occur in the SPMF or MPSF without adding any noticeable slowdown, and so at least for up to 500 jobs we can conclude that they are negligible in isolation, but have a quadratic effect on the per-job time when they both increase.

The performance of the SPSFS test was roughly 100 times slower than the sequential `sbatch` test. This is down to three key overheads, settling, querying, and executing. Firstly, each event has a 'settle time', where to prevent the `WorkflowRunner` getting swamped by multiple events from the same location any events that occur at the same location and very close in time are represented as a single event. This means after any single event the *File Monitor* process will wait for one second to catch any subsequent events at the same location. By definition, this will add at least one second of overhead to processing each event. This will occur in all tests, but in all others it will occur only once no matter how many jobs will be eventually scheduled, while in the SPSFS test it will occur for each sequential job. There is also the additional overhead of waiting for each job to be executed in turn. Aside from the raw time taken to execute the code, which is more complex in the `WorkflowRunner` jobs than in the Slurm testing, there will also be a delay inherent in the *Worker* process requesting a job from the *Queue* process. In these tests the *Worker* was set to query for new jobs every second. From all this we can conclude that in these tests at least the overhead from each of these was roughly a single second, and totalled 3 seconds. This is significantly slower that the approximately 0.035s achieved by the sequential `sbatch` test. However, it is worth highlighting that was in the case of the SPMF, MPSF and SPSFP tests, the SPSFS demonstrates very good scalability as it has a constant per-job overhead of approximately 3s. This overhead should remain constant even across a larger system of multiple users operating MEOW analysis at the same time.

E. Overheads on the MiG

Exactly the same tests were run on the MiG as on the `WorkflowRunner`. As with the other tests, the code and results are available at [14]. As can be seen in Figure 9,

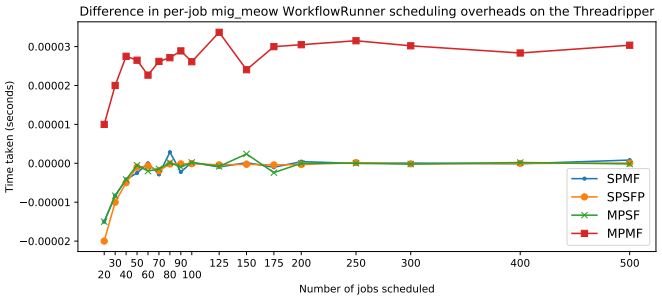

Fig. 8. Delta in per-job WorkflowRunner scheduling durations on the Threadripper. Note that the SPSFS result have been excluded as it has a much greater scale that crushes the rest of the results.

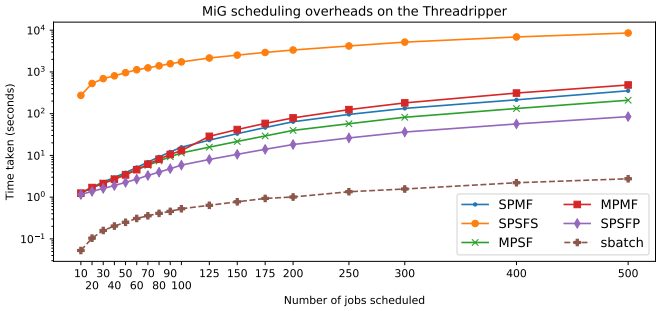

Fig. 9. Logarithmic MiG scheduling durations on the Threadripper. Each result is an average of 10 runs. The Slurm *sbatch* has also been shown as a dashed line for comparison.

performance from these tests is worse across the board, but this is to be expected. The MiG is a complete grid management system rather than a lightweight scheduler, so will always run much slower than either Slurm or the `WorkflowRunner`. With this in mind we shouldn't be too surprised that generally the MiG is roughly 25-50 time slower than `sbatch`. The sequential test is even slower at roughly 500 times slower than the sequential `sbatch` test. It is worth noting that whilst the SPSFS test demonstrates linear scalability, all others were quadratic in nature. This is demonstrated in Figure 10 where we can see that the difference in per-job timings is linear as the total number of scheduled jobs increases, which results in a quadratic increase in total scheduling time.

These broad increases are due to the nature of event processing on the MiG. As each event occurs the `watchdog` monitor will catch the event by performing some initial processing such as attaching a timestamp to it, and then matching it against the current list of *Rules*. This 'pre-processing' is kept to a minimum, and so any actions from a *Rule* match being carried out in a threaded function that must wait for processing resources to be available. This is done to keep the monitor process free to catch further events, but means that if many events happen at once then many different threads will be started. On the MiG this 'pre-processing' is quite extensive, with a number of authentication and robustness checks needed to be carried. Matching has been sped up through the use of regex, but the overheads are unavoidable. As many different matching events occur at the same time in these tests, more and more overheads are added by starting so many different

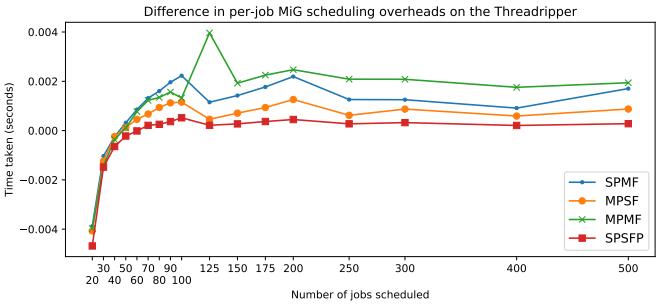

Fig. 10. Delta in per-job MiG scheduling durations on the Threadripper. Each result is an average of 10 runs. Note that the SPSFS result have been excluded as it has a much greater scale that crushes the rest of the results.

threads at the same time. Note that this does not occur on the `WorkflowRunner` as this uses the multiprocessing design does not require starting a new thread for each event.

Whilst it was expected that the MiG would run slowly, it was not expected that it would run quite as slowly as it has. Nevertheless, when we look at the results for the `WorkflowRunner` we can at least conclude that this is down to underlying issues within the MiG rather than with the MEOW system itself. as the same lack of scalability is not evident in the `WorkflowRunner`.

F. Evaluating the MEOW overheads

In light of all these tests we can conclude that MEOW does not add significant overhead to the scheduling process. It can be used to schedule large amounts of jobs at once, with a minimum of user definitions. This scheduling will demonstrate linear scaling when either a large amount of files or *Patterns* are in use at one moment. However, when both are in use at the same time the system can begin to experience greater and greater overheads. This is especially true in the MiG implementation. Even in the slowest non-sequential test (500 Laptop MPMF jobs) it must be remembered that only 0.97s was spent on scheduling each job and most tests ran considerably quicker than that. It does not seem unreasonable to expect that most scientific analysis would take considerably longer than this to complete, and so even this slow scheduling would quickly fade into the background compared to any execution time.

Although the scaling on sequential jobs is linear, a considerable amount of time can be spent before all required jobs are scheduled, as by definition all but one of them must be executed before they will all have been scheduled. We also see ever increasing overheads in the MPMF test. This is potentially concerning if MEOW were to be used as a central controller, such as in a large grid system like the MiG with many unique events occurring and being compared against many unique *Patterns*. If a sufficiently large enough number of users were using the system to schedule MEOW analysis, such that several hundred events were supposed to be triggering several jobs within the same few second then each user would begin to see increased overheads in line with the results presented in Figure 9. However, the proposed event driven systems is intended as a distributed one, with many schedulers on many

different resources each with only a few users at any one time. At the same time, the majority of scientific workflows do not produce masses of small files, but few large ones. Therefore, though this poor scalability from the MPMF test remains noteworthy, it should not be applicable the majority of MEOW use cases. Even with several users potentially creating many *Patterns and Recipes*, and producing numerous files at different locations, MEOW systems are capable of automatically scheduling analysis in trivial times at scale, especially compared to the hours or even days sometimes experienced in scientific processing. Regardless of the job processing time, MEOW can be expected to perform faster than Slurm with roughly a 2.5x speedup. As Slurm is already a widely adopted tool, this is an acceptable benchmark to pass and so we can conclude that MEOW has an acceptable level of performance in its scheduling.

IV. A Self-Modifying Workflow

A simple scientific example of a MEOW workflow has already been presented within [23] and so instead here we will examine a more interesting example of the new workflow structures made possible by the bottom-up design. Namely, the ability for a MEOW system to be self-modifying, and construct, modify or remove MEOW constructs at runtime. This is possible to do on both the MiG and within the new `WorkflowRunner` and so will be presented using both.

A. Problem Outline

In this example a user wishes to apply a filter to image data. However, the required filter regularly changes, even though the fundamental process does not. The users needs can be met with MEOW, by designing a system that will take configuration inputs to create new *Patterns*. Each of these *Patterns* will apply different filters to different data, according to their configuration. In this system the user only needs to manually write a single *Pattern*, and any subsequent requirements will be met by the MEOW system itself. This problem will therefore demonstrate how we can construct new MEOW *Patterns* from within a MEOW system to fundamentally alter the structure of the workflow. While *Recipes* perform the actual analysis, assembling a Jupyter notebook programmatically has been demonstrated numerous times before [6], and so will be overlooked here but is of course just as possible.

What we will create is a single *Pattern* and *Recipe*, resulting in one *Rule*. This *Rule* will schedule jobs to construct new *Patterns*, based on user provided configurations. The structure for this system is shown in Figure 11. Here we can see that our single *Rule* will respond to any configuration files placed in the *confs/* directory. This *Rule* will trigger jobs that will construct new *Patterns*, which will monitor different locations for data. These subsequent *Pattern* will create further *Rules*, that then schedule jobs as would be expected in any other MEOW *Rule*.

B. Predefined Patterns and Recipes

Before we define a Jupyter notebook for assembling new *Patterns*, let us describe the *Recipe* which the assembled *Patterns* will use. This is the *filter_recipe.ipynb* Jupyter notebook

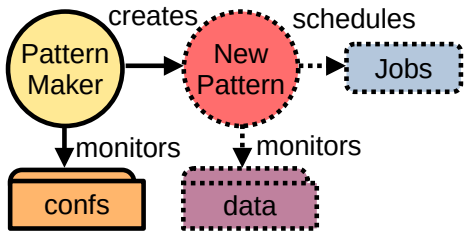

Fig. 11. The structure of the self-modifying example.

and in it image data is read in, a filter is applied to this image, and the filtered image is saved as a new file. The key part is that the filter command is created from the arguments provided to the Jupyter notebook. A valid filter command will be any of the filters available as part of the Python `ImageFilter` module [20], part of `Pillow` [32].

To construct new *Pattern* that will use the *filter_recipe.ipynb Recipe*, we will need a second Jupyter notebook. This is defined in *pattern_maker_recipe.ipynb*. This notebook reads in a configuration YAML file with then contents then parsed and used to construct a new `Pattern` programmatically. Once this new `Pattern` is complete, it is written to a specified directory. This is the directory where the `WorkflowRunner` will store the MEOW constructs, and is monitored by its *State Monitor* process. By writing a new `Pattern` directly to this location, we can insert it directly into the state of the `WorkflowRunner`.

At the start of the experiment, only a single `Pattern` is defined. This defines the directory path to be monitored, which in this case is just the *confs/* directory. All *Pattern* and notebook definitions, along with all input and output data files for both experiments can be seen in [14].

C. Using `WorkflowRunner`

A script was created to construct the necessary *Pattern* and *Recipes*, and then start a `WorkflowRunner` instance with them. Of note is that the internal state directory used by the `WorkflowRunner` is manually set to the same directory as the general file base directory, which in this example will be the *self-modifying/* directory. This means that both the State Monitor and File Monitor processes will be monitoring the same directory. Putting the state directory in the same place as the base file directory makes it easiest to access from within the jobs, which makes updating the `WorkflowRunner` state easier. This does mean there will be two monitors listening to the same file structure, so there may be some slowdown in responding to events due to each event being caught and processed twice. For a small proof-of-concept example like this on a local system, this will not create a significant overhead, though the problem will become more pronounced if used on something larger such as the MiG, and so is not generally advised.

When the `WorkflowRunner` is initially created, no data is present within the *confs/* directory, and so no *Pattern* creation jobs are scheduled. A user can also easily see the *Patterns* and *Recipes* that are registered in the system as they will each

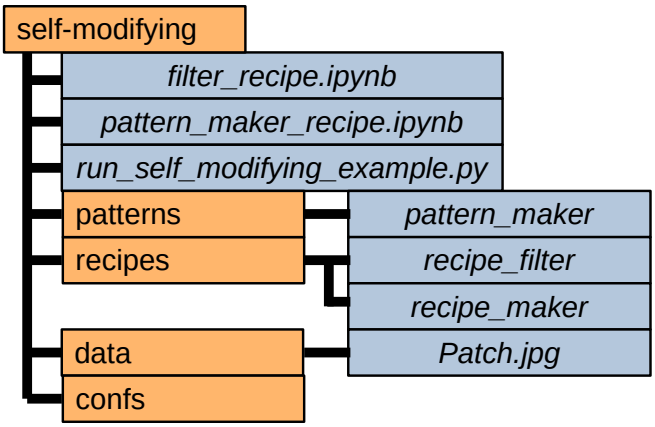

Fig. 12. The file structure of the self-modifying example before any configuration files are added to the `WorkflowRunner`. Directories are shown in orange and files in blue.

Fig. 13. Comparison of the input and output *Patch.jpg* data, used in the self-modifying example. Input data from *data/Patch.jpg* is shown on the left, with output data at *GaussianBlurred/Patch.jpg* on the right.

be in a *patterns/* and *recipes/* directory within *self-modifying/*. Before we start initiating jobs, we can add some image data in a *data/* directory. This will give us the overall structure shown in Figure 12. As no new *Patterns* have yet been created no processing can take place, even if the `WorkflowRunner` is running. To start scheduling some analysis we will need a configuration file to place in the *confs/* directory. Such a file is shown in Listing 1. This file is a YAML file containing a number of variable definitions, which match up to the expected inputs in the *pattern_maker.ipynb* Jupyter notebook. If this file is placed into the *confs/* directory, then the *Rule* created by *pattern_maker* will trigger, and a job will be scheduled.

```
input_path: data/*.jpg
output_path:
    '{VGRID}/GaussianBlurred/{FILENAME}'
filter: GaussianBlur
args:
    radius: 2
```

Listing 1. *input.yml* file contents.

This newly scheduled job will use the parameters specified in *input.yml* to create a new `Pattern`, which will be given the name *GaussianBlur_radius_2*, and will use the `GausianBlur` image filter in any resultant jobs. This *Pattern* will be saved into the *self-modifying/patterns/* directory, and so picked up by the `WorkflowRunner` for it to create a new *Rule*. As the *Rule* derived from this `pattern` will monitor the `data/` directory, and we have already placed an image file, *Patch.jpg* in said directory, a job will be immediately scheduled. In accordance with the already presented definitions, this will produce output which will be saved into the *GaussianBlur* directory. The sample input and output data used in this example are shown in Figure 13. As the `WorkflowRunner` is a continuous system, it will continue to monitor for additional events until the user stops it. Therefore we could add more data files, in which case more filter jobs will automatically be scheduled. Alternatively we can also add more configuration files to the *confs/* directory, which would create new *Patterns* and then lead to more filter jobs. If any of the configuration or data files were changed then the system would automatically reschedule the analysis without a user needing to restart anything. If any incorrect configuration were added, such as asking for a filter that did not exist, then naturally that job would fail. However that job would fail in isolation and all others could continue with no additional user input. This should demonstrate the utility of MEOW as a tool for scheduling jobs, in a robust, scalable and dynamic manner, where small parts or even the entire system can be changed at runtime.

D. Using the MiG

This same example was also carried out on the MiG, showing that although more difficult to achieve, the same results are possible even on a more restricted MEOW implementation. We cannot simply manipulate the MEOW state storage location on the MiG, as this part of the system is entirely hidden from the user. This is an intentional security feature, so that users do not manipulate data they do not have access to, or corrupt the state of a live system.

We can gain some limited access however by manipulating the mechanisms exposed in the inbuilt JSON messaging used in the `WorkflowWidget` [23]. To do so, we need to define some additional variables used within the MiG. Therefore we will update the old *pattern_maker.ipynb*, to a new *pattern_maker_mig.ipynb*. Mostly it is the same as before, but with the addition of the *WORKFLOWS_SESSION_ID*, *WORKFLOWS_URL* and *workgroup* variables. These will all be used in the modified *pattern_maker_recipe.ipynb* Jupyter notebook so that it can communicate directly with the MiG. As the *WORKFLOWS_SESSION_ID* is a security feature within the MiG, it has not been shown in the examples files in [14], but was of course used in the actual example run. The only other significant difference is that rather than writing directly to the state directory, we must instead communicate remotely with the MiG via JSON requests. In practice, this results in few code changes however, with locals writes being replaced with remote messages being sent.

As expected, once the `Pattern` and `Recipes` have been registered with the MiG, no jobs were scheduled until a file was added to the *confs/* directory. The same data file was added as before, and the *Pattern GaussianBlur_radius_2* was created on the MiG. This itself did not schedule any processing until the file *data/Patch.jpg* was added, at which point a second job was scheduled. This produced identical results to those shown in Figure 13, and so will not be repeated again here.

This is not an ideal solution however, as it depends on exposing security features of the MiG. This problem is limited in scope, as in order for a user to get to this stage they will need to have access to the MiG in order to spawn a Jupyter notebook with the necessary *WORKFLOWS_URL* and *WORKFLOWS_SESSION_ID* values. Therefore, as long as users are sensible with these credentials they will not be exposing data that would otherwise be secure. It is suspected that drawing attention to these variables which are otherwise hidden may encourage users to share them and so exacerbate the problem.

From a usability perspective though, the pressing problem is that *Patterns* and *Recipes* created in this manner will only be possible for as long as the *WORKFLOWS_SESSION_ID* remains valid. This is as only a *WORKFLOWS_SESSION_ID* will ever be registered for a user, and they will be regenerated throughout the lifetime of the MiG. However, creating a new *WORKFLOWS_SESSION_ID* will not update the variables passed in these *Patterns* and so when the resultant jobs try to send a message to the MiG they will be rejected. For this reason this solution is not suggested as a final implementation, but merely as a stop-gap demonstration of potential future functionality. A more robust implementation would be additional functionality within the MiG such as each job being assigned its own *WORKFLOWS_SESSION_ID*, thus allowing *Patterns* and *Recipes* interactions to be verified without having to share hard-coded credentials across jobs. A similar system is already in place on the MiG for SSH users within jobs, which allows individual job mount requests to be similarly authenticated, so it is not expected to be a significant challenge to do that same for MEOW interactions.

E. Concluding The Self-Modifying Example

Although this was somewhat of a toy example, with a simple configuration file taken as input, this is not the limit of the possibilities. Any input file could be taken in and parsed so as to produce new or modified MEOW constructs. Although only the dynamic creation of *Patterns* has been shown, it is perfectly possible for new *Recipes* to be constructed at runtime in the same manner, it would just take considerably more lines of code to create a new Jupyter notebook from scratch.

It is worth noting that we are not limited to merely adding new constructs, but can modify existing ones if we used some of the functions included in `mig_meow` for reading the current state of the workgroup. Here we could read in definitions, and write modified values back in the same manner as if we were altering them programatically within a Jupyter notebook. This means that a MEOW system can create, modify and delete itself, or its parts at runtime. It can also make decisions about when to do so within a suitably written *Recipe*, and so we can conclude that MEOW analysis is Turing complete at runtime. Whilst it would be bold claim to state that this is unique, none of the currently encountered SWMS have come close to this level of self modification.

V. Conclusions

This paper has explored event-based processing as a means for scheduling workflows in a dynamic and distributed manner. To better enable this a robust new tool has been introduced, the `WorkflowRunner` within `mig_meow`. Five benchmarks for event-based scheduling systems have also been described. These benchmarks demonstrate that event-based scheduling does not add significant overhead compared to traditional centrally controlled scheduling. Additionally, a short example is presented demonstrating how event-based systems can be so dynamic as to completely rewrite their own structure at runtime. This is something far more difficult to achieve in traditional top-down scheduling systems and so allows for new possibilities in workflow construction. This is expected to be of particular worth to distributed analysis systems, or in extremely heterogeneous systems accommodating human-in-the-loop interactions.

References

[1] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: An Extensible System for Design and Execution of Scientific Workflows. In: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004. IEEE (2014)

[2] Babuji, Y., Woodard, A., Li, Z., Katz, D.S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J.M., Foster, I., Wilde, M., Chard, K.: Parsl: Pervasive Parallel Programming in Python. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. p. 25–36. HPDC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3307681.3325400, https://doi.org/10.1145/3307681.3325400

[3] Badia, R.M., Conejero, J., Diaz, C., Ejarque, J., Lezzi, D., Lordan, F., Ramon-Cortes, C., Sirvent, R.: Comp superscalar, an interoperable programming framework. SoftwareX **3-4**, 32–36 (2015). https://doi.org/https://doi.org/10.1016/j.softx.2015.10.004, https://www.sciencedirect.com/science/article/pii/S2352711015000151

[4] Benedyczak, K., Schuller, B.T., Sayed, M., Rybicki, J., Grunzke, R.: Unicore 7 — middleware services for distributed and federated computing. pp. 613–620 (07 2016). https://doi.org/10.1109/HPCSim.2016.7568392

[5] Berthold, J., Bardino, J., Vinter, B.: A Principled Approach to Grid Middleware: Status Report on the Minimum Intrusion Grid. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) Algorithms and Architectures for Parallel Processing, pp. 409–418. Springer (2011)

[6] Creating an IPython Notebook programatically. https://gist.github.com/fperez/9716279 (2016)

[7] cwltool. https://github.com/common-workflow-language/cwltool (2021)

[8] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6. p. 10. OSDI'04, USENIX Association, USA (2004)

[9] Deelman, E., Blythe, J., Gil, Y., Kesselman, C.: Pegasus: Planning for execution in grids. Tech. Rep. Technical Report 2002-20, GriPhyN (2002), http://pegasus.isi.edu/publications/ewa/pegasus_overview.pdf

[10] Dias, J., Guerra, G., Rochinha, F., Coutinho, A.L., Valduriez, P., Mattoso, M.: Data-centric iteration in dynamic workflows. Future Generation Computer Systems **46**, 114–126 (2015)

[11] Docker. https://www.docker.com/ (2021)

[12] Dreher, M., Peterka, T.: Decaf: Decoupled dataflows for in situ high-performance workflows. Tech. Rep. ANL/MCS-TM-371, Argonne National Laboratory, Lemont, IL (7 2017)

[13] EuXFEL. https://www.xfel.eu (2022)

[14] Experiment results. https://sid.erda.dk/sharelink/F0hrjh3aEN (2022)
[15] Fahringer, T., Jugravu, A., Pllana, S., Prodan, R., Junior, C., Truong, H.L.: Askalon: a tool set for cluster and grid computing. Concurrency and Computation: Practice and Experience **17** (02 2005). https://doi.org/10.1002/cpe.929
[16] Ferreira, J.E., Wu, Q., Malkowski, S., Pu, C.: Towards flexible event-handling in workflows through data states. In: 2010 6th World Congress on Services. pp. 344–351 (2010). https://doi.org/10.1109/SERVICES.2010.60
[17] Foster, I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: IFIP International Conference on Network and Parallel Computing. pp. 2–13. Springer-Verlag (2006)
[18] Harenslak, B.P., de Ruiter, J.R.: Data Pipelines with Apache Airflow. Manning, 1 edn. (2021)
[19] Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (1978)
[20] ImageFilter. https://pillow.readthedocs.io/en/5.1.x/reference/ImageFilter.html (2022)
[21] Kubernetes. https://kubernetes.io/ (2021)
[22] Marchant, D.: mig_meow on PyPi. https://pypi.org/project/mig-meow (2021)
[23] Marchant, D., Munk, R., Brenne, E.O., Vinter, B.: Managing event oriented workflows. In: 2020 IEEE/ACM 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP). pp. 23–28 (2020). https://doi.org/10.1109/XLOOP51963.2020.00009
[24] Mattoso, M., Dias, J., A.C.S.Ocaña, K., Ogasawara, E., Costa, F., Horta, F., Silva, V., de Oliviera, D.: Dynamic steering of HPC scientific workflows: A survey. Future Generation Computer Systems **46**, 100–113 (2015)
[25] MAX IV. https://www.maxiv.lu.se (2022)
[26] Montella, R., Di Luccio, D., Kosta, S.: DagOn*: Executing Direct Acyclic Graphs as Parallel Jobs on Anything. In: Proceedings of WORKS 2018: 13th Workshop on Workflows in Support of Large-Scale Science, Held in conjunction with SC 2018: The International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 64–73 (2019). https://doi.org/10.1109/WORKS.2018.00012
[27] Munk, R., Marchant, D., Vinter, B.: Cloud enabling educational platforms with corc. In: Proceedings of the 8th Workshop on Cloud Technologies in Education (CTE 2020) (2020)
[28] notebook-parameterizer. https://pypi.org/project/notebook-parameterizer/ (2021)
[29] Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17), 3045–3054 (2004)
[30] Oracle Grid Engine. https://www.oracle.com/technetwork/oem/host-server-mgmt/twp-gridengine-overview-167117.pdf (2010)
[31] papermill. https://github.com/nteract/papermill (2021)
[32] Pillow. https://pillow.readthedocs.io/en/stable/ (2021)
[33] Ramon-Cortes, C., Lordan, F., Ejarque, J., Badia, R.M.: A programming model for hybrid workflows: Combining task-based workflows and dataflows all-in-one. Future Generation Computer Systems **113**, 281–297 (12 2020). https://doi.org/10.1016/j.future.2020.07.007, http://dx.doi.org/10.1016/j.future.2020.07.007
[34] Rocklin, M.: Dask: Parallel computation with blocked algorithms and task scheduling. In: Proceedings of the 14th python in science conference. pp. 126–132 (01 2015). https://doi.org/10.25080/Majora-7b98e3ed-013
[35] Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R.M., Torres, J., Cortes, T., Labarta, J.: PyCOMPSs: Parallel computational workflows in Python. International Journal of High Performance Computing Applications **31**(1), 66–82 (2017). https://doi.org/10.1177/1094342015594678
[36] Tolosana-Calasanz, R., Bañares, J., Álvarez, P., Ezpeleta, J.: On interlinking of grids: A proposal for improving the flexibility of grid service interactions. In: Proceedings of The Third International Conference on Internet and Web Applicaations and Services: ICIW 2008. pp. 714–720 (07 2008). https://doi.org/10.1109/ICIW.2008.39
[37] Torque Resource Manager. https://adaptivecomputing.com/cherry-services/torque-resource-manager/
[38] Voetmann, N.A.T.: Framework for Uploading Research data (FUR). Master's thesis, Niels Bohr Institute, University of Copenhagen (3 2021), https://sid.idmc.dk/sharelink/hZjJ5NSvlb
[39] watchdog. https://pypi.org/project/watchdog/ (2021)
[40] Workload Manager. https://www.ibm.com/docs/en/aix/7.2?topic=management-workload-manager (2021)
[41] Yildiz, O., Ejarque, J., Chan, H., Sankaranarayanan, S., Badia, R.M., Peterka, T.: Heterogeneous hierarchical workflow composition. Computing in Science Engineering **21**(4), 76–86 (2019). https://doi.org/10.1109/MCSE.2019.2918766
[42] Yoo, A., Jetter, M., Grondona, M.: Slurm: Simple linux utility for resource management. Lecture Notes in Computer Science **2862**, 44–60 (2015)