Kalpa Publications in Computing

Volume 17, 2023, Pages 15–30

Proceedings of 2023 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

Slurm Scheduling From Rules-Based Systems

Mark Blomqvist[1] and David Marchant[2]

[1] University of Copenhagen, Copenhagen, Denmark cvp375@alumni.ku.dk
[2] University of Copenhagen, Copenhagen, Denmark david.marchant@di.ku.dk

Abstract

This paper explores how a rules based scheduling system can be integrated with a traditional workload manager such as slurm. This integration will be done with as minimal additional setup by a user as is feasible, while maintaining security of the network, as well as any machines involved. To this effect the processing component known as the Conductor in the rules-based scheduling system MEOW, has been extended with a remote option. This will enable MEOW to transmit jobs to a remote system, with the option of using slurm to orchestrate them. The new option is evaluated by comparing the execution time of using the remote solution without slurm, with that of existing components. Furthermore, the overhead of using various slurm methods for scheduling jobs with that of running the jobs remotely without using slurm is compared. It is shown that the remote solution adds a flat overhead to the execution time as both the number of jobs and size of the transmitted data increases, and that the overhead associated with using slurm on top of it is largely insignificant. This is considered to be an acceptable result given the test environment that was used.

1 Introduction

Scientific analysis has a recognised need to be dynamic, and this is not well met by existing static workflow systems[17]. Such systems, as typified by Apache Airflow [10], Pegasus[7] or Kepler[3] used Directed Acyclic Graphs (DAGs) to structure and schedule analysis in a robust, but difficult to adapt manner. Alternatives to this do exist, with one prominent possibility being a rules-based processing. This is a system where events are matched against user-defined rules, with matches resulting in some effect. Exactly what events, rules and effects are differs by system, with examples including Ripple[5], SPADE[1], SPOT[8] and Triggerflow[12]. Though these examples differ in nature, they typically do not schedule complete workflows and are instead limited to trivial tasks such as moving files around or cleaning data files. Where more complex analysis is triggered, it is done by starting a complete linear workflow.

A more complex rules based systems is Managing Event Oriented Workflows(MEOW)[15]. This is a structure for defining rules which can match broad categories of events and can schedule arbitrary analysis in response. This analysis is completed in isolation to any other, and may trigger further analysis. It is this potential chain of analysis which will form a workflow that is both robust to errors, can form novel analysis structures such as branches and loops, and can easily accommodate human-in-the-loop interactions. MEOW implementations have relied

on multi-process system to receive events, schedule jobs, and complete analysis, but have been shown to not add unreasonable overhead to scientific analysis[13]. However, the system has been limited by a lack of integration with existing scheduling systems. This may act as a barrier to adoption, but also means that optimisations made within those systems would need to be re-implemented within MEOW, taking up time and resources that could be better spent elsewhere. This paper will describe an extension to `meow_base`, a Python implementation of MEOW[6]. The extension consists of a remote option for the conductor, to enable the transmission of jobs to a remote resource for processing.

1.1 MEOW

MEOW is a work-in-progress dynamic scheduler that utilizes events as a means of dynamically creating a workflow. The core of MEOW are the concepts of recipes, patterns and rules. A recipe is a block of code that is used for defining job processing. A pattern is the conditions under which a recipe is processed and defines when to schedule the potential job. In other words it determines which events should produce jobs. Apart from a criteria for events, a pattern also points to the recipe that should form the job, as well as paths for input and output files. One can think of this as an if-then relationship, with the if statement being the pattern and the then statement being the recipe. Both recipes and patterns are defined by the user. This means that a user could define a pattern for a recipe which is never defined, in which case nothing happens. This is by design such that we never try to match events against a pattern for which no recipe exists, which is a waste of time. If a pattern matches an existing recipe it will form a rule. A rule is simply put the amalgamation of an existing pattern and existing recipe, and this is then checked against incoming events. If an event matches the conditions of the rule, one or more jobs are scheduled.

Initially MEOW was developed to run on the MiGrid(Minimum intrusion Grid)[2], University of Copenhagen's grid solution, in a project called `mig_meow`[14]. A more generic version has since been created called `meow_base`[6], that is not tied to using the MiGrid when running workflows. This was to allow for future implementations, as the MiGrid does not reflect the setup of all possible HPC grids.

In `meow_base` a so called `WorkflowRunner` handles the complete lifetime of any and all event handling and job processing. It is comprised of several components, each running as a separate thread or process. These are conductors, handlers and monitors, which will each manage a particular point in MEOW job lifetime. A monitor is created to check for any events, and to match them against any user defined rules. Currently only **FileEvents** and **NetworkEvents** are supported ie. a file has appeared in the directory, or data has been sent from a remote network location. When the monitor detects an event, and it matches a rule, it will send it to the handler queue. Any handlers will poll this queue, and then examine a given event to determine if it can handle the job. If this is the case it will create a job script and submit the job to a job-queue. A job-script is a bash script that is primarily is used to execute the recipe, as well as check if the input data has changed, but more on this later. The conductor will poll this queue, and determine if the job in the queue is of a type it can process. If so, then it will proceed to execute it. This is where the extension has been implemented, through the addition of an additional argument passed to the conductor that determines if the job is to be executed on a remote resource running slurm. Additionally the slurm-specific arguments can be passed as a second additional argument to the conductor. An illustration of the workflow runner can be seen in Figure1

This structure for the `WorkflowRunner` uses multiple processes in order to efficiently run all

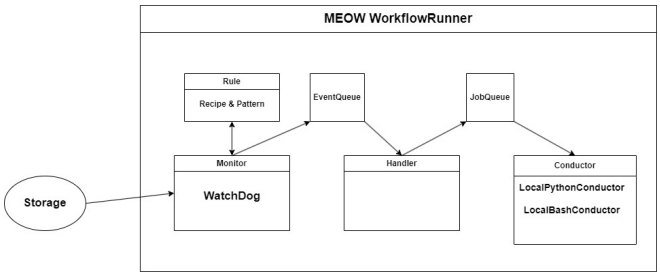

Figure 1: The MEOW WorkflowRunner. Some parts have been left out for simplicity. Note that the named conductors, ie. python and bash, are started as instances of the conductor.

components. A reliable system cannot miss events, and should be able to process them as quickly as possible. Therefore, it has been separated out from other system components into its own process so that events can be processed without job creation and processing getting in the way. Each monitor, handler and conductor is its own process, and so can fail, be delayed, or added at runtime as is necessary, without compromising the system as a whole. This is in keeping with MEOWs adaptive design philosophy, though adds a danger of concurrency problems. To address this, the design was created according to the principles of CSP(Communicating Sequential Processes)[11]. According to this, as no loop of reader/writer dependencies exist, no deadlock can occur. Similarly, no data is shared between processes and so no race-conditions can occur. We can therefore conclude that the system is concurrency safe.

1.2 Remote Processing Systems

Existing HPC grids usually have an existing solution installed for scheduling and processing tasks. These solutions have many desired features that could be of use if MEOW where to be integrated into these, instead of having to re-implement them into MEOWs conductor. Therefore this project will integrate MEOW into one of these solutions, namely slurm.

Slurm is an open-source cluster-management and job scheduling system for Linux clusters, in other words a workload manager. The relevant parts of slurm in the context of this paper will be the controller daemon **slurmctld** and the compute daemon **slurmd**. In its most basic form a single centralized controller(with the possibility of a backup controller), will oversee a various number of compute nodes, each with the slurmd daemon running on each compute node. Setups with multiple controllers are possible, but for the duration of this paper, the controller with be referred to as a singular entity. There is also a login-node which acts as an intermediate between the user and controller. The controller monitors the compute nodes and their available resources as well as the work being executed on them. Each compute node is grouped into a partition, of which a job can be granted a subset of this partition. A job can then be split into a step and allocated on a node, which can run in parallel if needed. The compute nodes simply waits for work, executes it and returns a status after which it awaits more jobs. A simplified diagram with the relevant parts is shown in Figure2. Since the hardware available for testing was limited we decided to have a single node acting as the login, controller and compute node with the option of an additional separate compute node. This setup was abstracted as the **Remote** as is illustrated on Figure3. Slurm has a variety of methods for executing jobs, of which two are relevant to this paper; **srun** and **sbatch**. Srun is used to execute a task in parallel and is blocking. It will create a resource allocation if necessary and executes the task

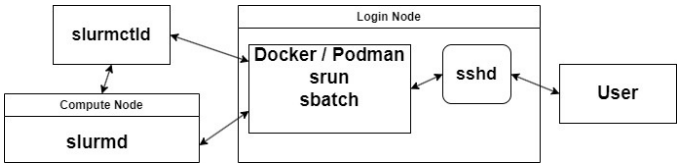

Figure 2: Basic setup of slurm[20]. Any entities not mentioned in this section, like munge-perimeter and slurmstepd, can be ignored, as they are not relevant for the content of this report.

as soon as possible. Sbatch is used to submit a batch script for later execution, and is therefore not blocking.

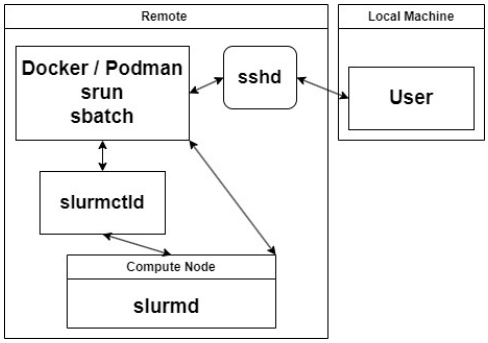

Figure 3: The test setup used for evaluation of performance. The controller- and compute daemon is running on the login node. The separate compute node is absent for simplicity

MEOW is intended to be used for exploratory scientific research, where the workflow cannot be determined before run-time. Since scientific research of this kind can be computationally demanding the processing is usually relegated to HPC(High-performance computing) grids. Therefore it will be easier to integrate MEOW with existing setups, if it can interact with one or more of these solutions. This project therefore aims to integrate slurm into MEOW. These clusters are not always available for direct in-house access, and thus the analysis data and subsequent results must be transferred to and from these clusters. The COVID-19 pandemic only increased the need for remote access to this infrastructure, and even though reliable statistics on the frequency and size of cybercrime are scarce[21], the need for remote users to follow security guidelines has only grown in importance[19]. In addition to moving the processing to a remote resource, this implementation will also attempt to protect both endpoints by using containers on the remote host and limiting the remote host's access to the users private directories with sand-boxing.

2 Implementation

In this section the sequence of events from the time the conductor receives a job from the local job queue, to the job being received in a finished state on the local machine will be outlined. The approach is to use prebuilt containers and then mount a local chroot sandbox through an sftp server(running on the local machine) inside this container(running on the remote resource).

Initially Docker was used, but was soon replaced with rootless-Podman as it was assumed to be more secure engine. Later a thesis was discovered that compared the attack surface between the two container engines[9], and it appears that the question is more complex than it initially seemed. Although rootless-Podman still wins on most parameters, the attack surface is reduced significantly more by using local images instead of downloaded ones. This implementation only uses locally built images, but for this implementation to be used at scale this is an issue worth considering, however that is out of scope of this paper. A Github page for hosting a copy of The meow_base code with our additions is hosted on Github[16]

2.1 Methodology

In order to make the final design easy to use, the system will be built so that as much as possible, it can all be operable from the users machine. Moreover we want the additions to make MEOW as portable and easy so integrate with minimal remote setup. Therefore the setup on the remote is as basic as possible ie. we only have slurm 23.02 installed, an open ssh-server with key-based authentication and a pre-built podman image. Rootless-Podman is also assumed to be installed and configured, and public keys are shared between the two machines. Though this may initially seem a lot, it is worth noting that this is all a one-time setup and that any subsequent user of the system could so do without having to do anything.

Another reason we want to make as few assumptions about the remote resource as is feasible, is that it is assumed that the remote is an untrusted system. This might be considered an unrealistic assumption, but we will use it as a guideline to limit what is shared between the local and remote machine. This is because we will aim to prevent the user from being able to use the system in an insecure way. Additionally we will aim to protect the remote resource by only executing the job inside containers. Given that the potential data transfer can be very large, we will not copy data back and forth directly, but instead use mounting for this purpose.

2.2 The conductor

The conductor has been modified such that we can choose weather to do the processing locally or on a remote resource. If remote processing is chosen the conductor will translate the job into a slurm job-script, such that it can executed by slurm on a remote resource. It does this translation on the basis of the metafile. Each job has one such metafile: a YAML-file that contains information about the job eg. the job-type and relevant timestamps like time of creation, start of execution and completion. An illustration of whole implementation can be seen in Figure 4.

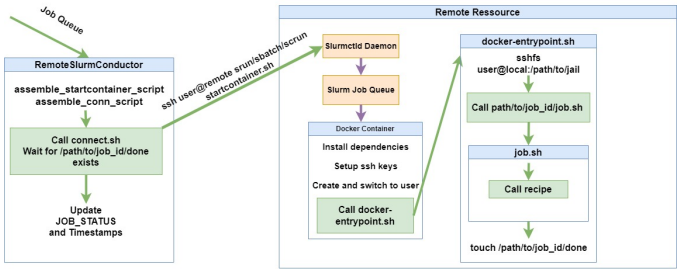

Figure 4: Structure of sending and executing a job on a remote resource using slurm

This diagram represents the sequential execution each time a conductor receives a job from the job queue. It is possible to not use slurm for testing purposes, and this will be used for some of the overhead tests.

When the conductor is set to execute the job remotely it will incorporate an additional optional arguments in the form of a string list containing the relevant slurm arguments. This is supposed to emulate what should be user defined, but isn't supported by MEOW at time of writing. They will be used to assemble **connect.sh** and **startcontainer.sh** if we want to use slurm methods for processing. If this argument is not set, the conductor will execute directly instead.

Firstly it updates the metafiles **tmp script command** entry to execute the connect.sh script instead of the standard **job.sh** created by MEOW. It then assembles(ie. not executes) connect.sh.

The purpose of this script is to connect to the remote SSH-server via key-based authentication and execute startcontainer.sh with either srun or sbatch. To assemble this it uses 3 arguments; the job directory for jobs it is supposed to process, the output directory for where it is supposed to store the output, and the aforementioned slurm arguments. The reason for this is that it needs to know if the user has specified sbatch, which will slightly alter the command we want to execute through our ssh-connection. Since sbatch is supposed to be used on a batch-script it accepts input from standard input, so it made the most sense to use sbatch here and then add the **#SBATCH**-arguments in startconinter.sh, instead of passing the arguments as options to sbatch.

If srun is present in the slurm arguments, or if we are executing it without slurm, we execute startcontainer.sh over ssh by redirecting the script to bash on the remote resource.

Then it assembles the startcontainer.sh script which contains the **podman run** command. In order to assemble, it needs to know the job id, the job directory and slurm arguments, if specified. Depending on what is given in the arguments it will write an appropriate call that will run our container with the arguments provided. In all cases the container is passed the current job id as an environment variable. The conductor will then execute connect.sh which will send the job to the slurm job queue. When the job has been submitted to slurm, the conductor will update the job status and at time of writing, as far as MEOW is concerned the job is completed. Ideally we would update the metafile inside the container when the recipe is done, but this has yet to be developed.

The slurm controller daemon will put the job in its own job queue and handle getting it to an appropriate execution node running the slurmd service. As mentioned this part has been simplified greatly as we have used a single node for login, controller and computation, but in reality these should be separate.

2.3 The Dockerfile

An image based on a Dockerfile has been created and is assumed prebuilt on the remote resource. The image installs the required dependencies, takes a copy of the required private key for the local sftp-server, configures **fuse.conf**, creates a directory structure that resembles the chroot sandbox on the local machine. Finally it uses **ENTRYPOINT** to run **docker-entrypoint.sh** which is also assumed to be pre-written and located on the remote. This docker-entrypoint.sh uses sshfs to mount a directory from the local machine, using key-based authentication. The reason for using key-based authentication is that we want the process to be automated, and

even though having both password- and key-based authentication is considered more secure, it is too impractical to write ones password for each job. As mentioned the container will be run in rootless-mode, but since FUSE requires some access to the kernel it currently cannot be avoided to add –**cap-add SYS_ADMIN** –**device /dev/fuse** as arguments to podman run. This has some security implications even for rootless-podman, as it might add attack vectors for privilege escalation within the container. A solution to this is to mount outside the container and pass the mount point as an argument to the container on startup, but we ideally only want to be mounted while a job is running.

The reason we want to mount/unmount on a per job basis, is that we can't be sure if more jobs will appear in the job queue even if it is empty, and would therefore have to be mounted even if no jobs were present.

The local machine is then running an ssh-server that only allows sftp connections such that the remote theoretically cannot get a shell on the local machine, and we can create a user account with limited permissions for only this purpose as well.

2.3.1 Executing the job

Once the sandbox is mounted we execute the job.sh from the job directory, which we can identify by the environment variable passed to the container when startcontainer.sh was assembled.

Finally job.sh will call the recipe. The path to the recipe and redirection of the output is determined by MEOW when its assembled. Once the recipe is done with its task, job.sh will return, and docker-entrypoint.sh will create an empty file called **done**" in the same directory as the other files for that job id, ie. the job-directory for that job. This file can then be checked by a local monitor. We do this as we only have a file-transfer protocol available for communication. At some point the monitor of MEOW should be updated such that it can listen for these files and report back when the workflow is done.The monitor in MEOW already uses inotify which adds insignificant overhead, but at time of writing this has yet to be developed. As for now we have timed it using UNIX's **watch** command, which is expected to have added some overhead to the execution times of the remote solution. This marks the end of a single job execution.

3 Test Results

Our test environment consists of a Lenovo Thinkpad E15 Gen2 which we will refer to as the local machine, and a desktop running one or more virtual machines, all of which we will refer to as the remote. Both are running Ubuntu 23.04 LTS. The virtual machine is running with 6 cores and 8 GB of memory, where the native system has an Intel Core i5-8400 @ 2.80GHz and 16 GB of DDR4 RAM. The local machine has an AMD Ryzen 5 4500U @ 2.3 Ghz and 8 GB of DDR4 RAM. While these specs vary slightly we will demonstrate in the following tests, that this difference does not significantly impact the test result. This setup runs on a local network with 1000/100Mbit, and both machines connected via Wi-fi. This is obviously not an ideal setup, as in practice the remote machine would not be on the same local network, and the desktop running the virtual machines does not reflect the hardware of an HPC. Therefore the transfer times will be significantly less than what is realistic, and the execution times will be significantly slower than is realistic. Moreover the recipes used in the tests will contain trivial computational work, thereby attempting to minimize the impact of the difference in the processors. Despite this it is still possible to demonstrate that we do not add significant overhead.

3.1 Overhead tests

The remote solution was tested for a variety of number of jobs, using the test for **SweptPythonExecution** found in meow_base/tests/test_runner.py as a baseline. Each job will read 1MB of transferred data and write 1 byte back. Each parameter is tested 5 times. This tests executes the container directly and therefore does not use slurm, but as will become apparant later this does not significantly impact the result for a single compute node. Table values as well as average overhead in percent per parameter can be found on the Github page under *testresults*, Table 3 and 4 and the recipe as Listing 1[16]. As is illustrated in the above figure

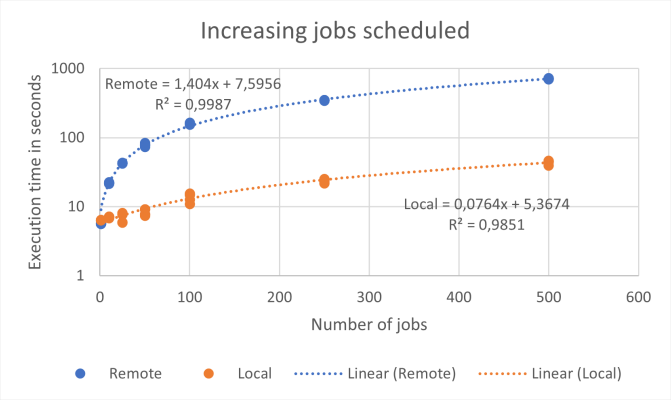

Figure 5: Comparison between remote and local processing for a variety of number of jobs with total execution time. Each job reads 1MB and writes 1 byte. The overhead scales linearly with the number of jobs

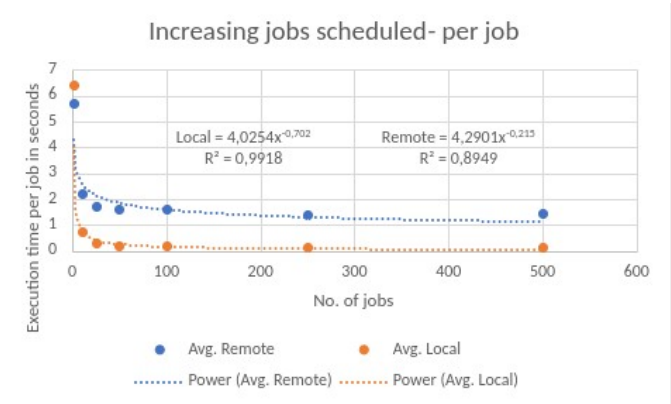

Figure 6: Comparison between remote and local processing for a variety of number of jobs with per-job execution time. Each job reads 1MB and writes 1 byte. The overhead scales linearly with the number of jobs

the execution times of the remote solution has a linear relationship to the number of executed jobs. Since the linear regression has an R-squared value of 0,99 for the remote example and 0,98 for the local, we will proceed with this as a good model for 10 or more jobs. Although

it is good practice to use more metrics in this scenario it is considered sufficient. The reason one should only use it on 10 jobs or more, is that it will not predict correct value below this threshold, certainly not for 1 job as it will claim 7,5956 even for 0 jobs. Using this model for at least 10 jobs we predict the per-job running time of around 1,4x+7,6s, where x is the number of jobs. Comparatively the local solution can also be modelled by a linear relationship with a per-job running time of around 0,08x+5,37. We can observe that the per-job execution time is slightly decreasing as the number of jobs increase. This means there is a flat overhead in the scheduling of jobs. There are a few suspected reasons for the overhead.

Before we get to that we will briefly touch on a small anomaly in the data. Although not immediately apparent from the figure, the table values show that the remote solution is faster than the local up to and including 5 jobs. The reason for this is suspected to be the faster clock speed of the remote machines CPU, which makes reading the 1MB faster. This is easily checked having the recipe sleep 1 second instead of reading. Again this recipe can be found on the Github as Listing 2[16]. For an average of 5 tests each it is clear from Table1 that the local solution is faster for 1 job, if we simply let it sleep 1 second.

1 job w. sleep 1	Execution time in seconds
Remote	7,32s
Local	6,022s

Table 1: Remote vs Local for 1 job that just sleeps 1 second and write 1 byte to output file

Therefore the main contributor for overhead is likely setting up the connection and connection retries, since ssh uses the TCP protocol and subsequent overhead associated with that. More over if the call to sshfs in the container fails because of connection reset, it takes a few seconds to retry. Usually 1 is sufficient, but 2 retries has been observed. This is fairly rare, but adds to the variance in results. In much the same way if the ssh command from the connect.sh fails because of a connection timeout, this adds a few seconds as well though this is a very rare occurrence. These occurrences were not timed, as we had no immediate solution for reducing or eliminating them, and they were hard to accurately time, without adding non-insignificant overhead. Starting the container also adds some overhead though this is probably minimal. In an real scenario we would expect the growth of the linear model to differ as distance between the local and remote machine will be greater and the remote will have more processing power, but we do not expect another model class to fit better than the linear one applied here.

3.2 File transfer test

Since the transfer of data files is suspected to add a lot of additional overhead a test was conducted that read the entire inputfile for various sizes for a single job. The recipe used for Figure5 is used for this test as well. Again these tests does not use slurm. The values used are averages, but a table for all datapoints can be found on the Github, as well as a table with averages and associated overhead in percent in Table5 and Table6.

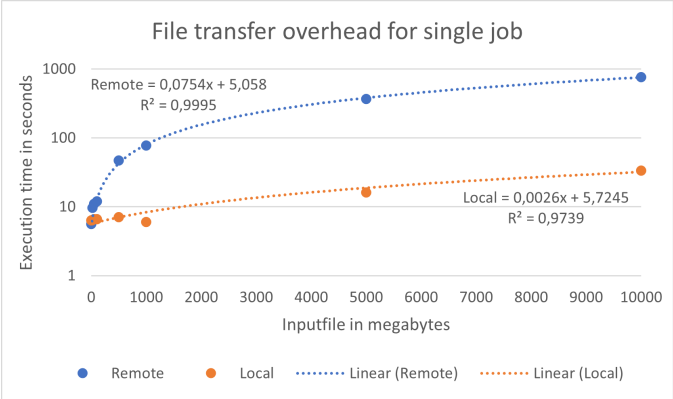

Figure 7: Comparison between remote(blue) and local(orange) for a single job with varying inputsizes. The model should only be used for file sizes at least 1GB or more.

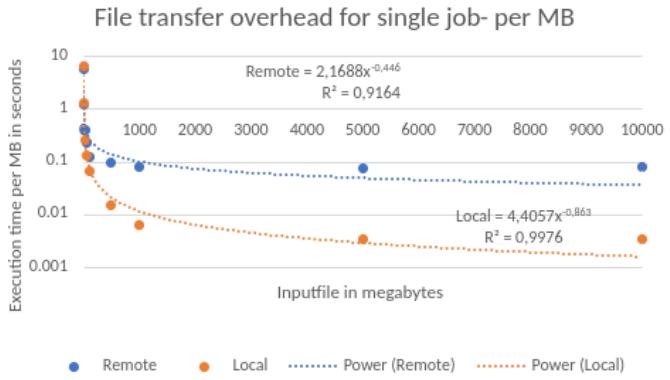

Figure 8: Comparison between my implementation(blue) and meow_base locally(orange) for a single job with varying inputsizes. The model should only be used for file sizes at least 1GB or more.

The entire input-file was read in chunks of 100MB at a time. It appears that there is also a linear relationship between the file size and execution time, which yields and execution time of 0,075x+5,06s where x is the number of megabytes. Again this is considered a good result, as it is linearly increasing. One anomaly can be seen for the local model, where the execution time drops slightly for 1GB. Since the remote solution is the focus here, we did not investigate this further, but it is assumed to be some optimization done by the local system.

3.3 Scaling with increasing jobs and inputsize

Since MEOW is supposed to be used for both multiple jobs and multiple inputsizes a test was conducted in order to determine how well my implementation scales for both a large input size and a high amount of jobs. Given the time these tests takes only a single test was made for each. The predicted values are calculated by plotting in the file size in the model obtained when using file sizes, and then just multiplying the number of jobs, without using the model

Implementation	25Job 1GB	100Jobs 1GB
Remote Actual	2025,1s	7550,89s
Remote Prediction	2011,45s	8045,8s
Local Actual	31,66s	114,38

Table 2: The actual execution time compared with a rough prediction. The actual values are very close to the prediction though.

associated with that. For example $(\mathbf{0,0754 * 1000 + 5,058}) * \mathbf{25 = 2011,45s}$. The reason for this is that both of the models contains some of the same setup, and so this rough estimation is used. However the actual values are slightly better than the predicted values. Even for a rough estimate we conclude from this that the implementations performance is in tune with the models and the execution time is predictable.

3.4 Slurm tests

Both srun and sbatch was compared with the direct solution for increasing number of jobs scheduled on a single compute node. To better demonstrate the benefits of using slurm over of the direct solution, an additional compute node was added. This is because slurm can manage work across multiple nodes, whereas the direct solution can only use a single node for computation. Moreover the conductor was changed such that connect.sh didn't wait for the done-file to appear, but simply executed the slurm-method, and updated the job status as soon as the command returned. Additionally rootless Podman was used for these dual-node tests, however they use the same Dockerfile as the previous tests. The same recipe used in Figure5 was used for all the tests and all file transfers are 1MB with 1 byte written back. No slurm arguments were used in these tests. As expected sbatch performs the worst for a single

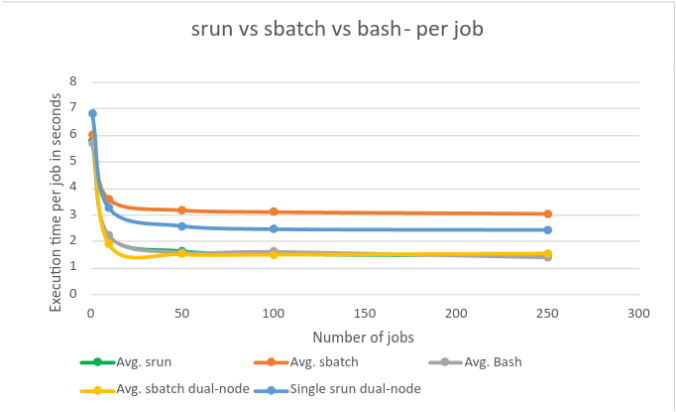

Figure 9: Comparison between srun, sbatch and running without slurm(bash). file size transferred is 1MB read and 1 byte write. Note both slurm methods were tested on a single and dual-node setup.

node setup, as it will background the job for later execution, but since this setup only runs sequentially this is just unnecessary busy work. The overhead of using srun is insignificant compared to running directly. On the other hand, using just one more compute node we get

the same scheduling time as running directly on a single node. Therefore it seems reasonable to assume that with just 3 compute nodes we will see sbatch outperform the direct solution. The dual-node test for srun was only ran once, as it became apparent that the controller daemon favored the newly added remote compute node, as opposed to the compute daemon running on the same machine as the controller. Therefore the performance of srun was slightly worse than when we only had 1 compute node. However we do observe a flat increase in the overhead, which the per-job execution time decreasing slightly as the jobs increase. The table values for each datapoint can be found on the Github as Table7 and an average in Table8[16].

3.5 Actual Scientific Work Test

A final test was conducted using the scientific analysis first presented in [15]. This is an example of a real-world scientific workflow, and is tested here to show that our results above still apply to actual analysis. As this workflow and its analysis is unchanged since that first benchmark, it will not be explained in detail here. A short summary is that the workflow analyses 2d scans of 3d foam structures. A variety of foams are taken as input, some which have too many bubbles, some which have too few and some which are just right. As this analysis is computationally expensive, we only wish to do it on the acceptable data and so the first workflow step is to pre-process the input data into acceptable or not. Unacceptable data is discarded and no further analysis is carried out, but acceptable data is segmented, analysed and a final report is assembled. The key part of this is that we do not know ahead of time which data is acceptable or not, or how much of it there is and so the workflow structure is adapted at runtime.

1st run, cold start	2nd run, warm start
Remote(Idle=45):	Remote(Idle=45):
722,01	604,2
Local(Idle=45):	Local(Idle=45):
641,22	640,49

Table 3: Execution times for actual scientific work.

As can be seen from Table3 we do in fact get a better performance for scientific work with the remote solution.

4 Discussion

As mentioned we need a monitor to keep track of how many done-files are present in the job directory. Additionally when this file exists it should move the contents to a dedicated output directory, as is MEOW behaviour locally. However, since we consider the job to be completed when it enters the slurm job queue, we can't put this behaviour inside the conductor. On top of that the monitor should determine when all "done-files" are present, such that it can communicate when the entire workflow is completed, given that we know how many jobs should be scheduled in the workflow. If we cannot be sure of how many jobs are to be scheduled, we cannot reliably tell the user when the workflow is complete, but most of these cases are suspected to be continuous workflows anyway.

4.1 srun vs sbatch

There are several things to consider when choosing a slurm method to run a job. On the one hand srun is ill-suited when used alone, because it is a blocking operation and thus doesn't allow for MEOW to simply submit the jobs and let the remote worry about the processing. Still for just one compute node we see srun outperform sbatch in the tests on Figure3, because with sbatch we background the job for later execution. However by adding just 1 more compute node we see sbatch run about as fast as using srun for 1 compute node and therefore it stands to reason that using sbatch to submit jobs to slurm will scale well as the number of compute nodes increases. Additionally we see that srun and is almost identical to running the container directly, and therefore using srun adds insignificant overhead for 1 compute node as opposed to running directly. Since they are all linearly increasing this adds largely insignificant overhead.

An immediate problem with running the workflow in parallel is race-conditions when reading the input data. Some solution could be explored for this issue, although this could be regarded as a feature, as some workflows continuously monitor the same input data and are meant to run endlessly[4].

4.2 Overhead test results

The test results shows a fairly linear relationship between the execution time and number of jobs. It also shows a linear relationship between the amount of data transferred and execution time. Therefore the implementation is best used with as few read/write commands to data files as possible, and instead utilize the HPC's hardware for computationally heavy tasks instead. We even see the remote solution outperform the local one slightly for just 2 compute nodes. This is somewhat the expected results, but the important thing to note is that the overhead increases more less linearly. Some anomalies exist, where the execution time is larger than expected, but these are mostly caused by connection timeouts and resets when mounting and using ssh to call startcontainer.sh from the local machine.

4.3 Alternative design

An alternative to this design is to drop the login-node and run the control daemon on the local machine, and then have the remote execution nodes connected to that instead. A potential problem with this design is that slurms compute daemons are controlled by a single controller, so how this is going to work for multiple users is at this point unclear. Furthermore if the users machine is compromised it will have local access to the controller daemon and through that all the compute nodes. The upside is that we will not need to first connect to a login node and through that connect to the controller, thereby eliminating more overhead. It will also make it easier to use containers, as the user can write and build them locally, without assuming they are prebuilt on the remote resource.

4.4 Future work

There are a lot of immediate additions that warrants further exploration. First of all we should update the job status inside the container as soon as the recipe is complete. This would also remove the need to create files for the monitor to look at, and instead make it read the metafile, but this might add overhead when conducting tests as one would need to read the metafiles for their status quite often. In a practical scenario one wouldn't need to read the status in the

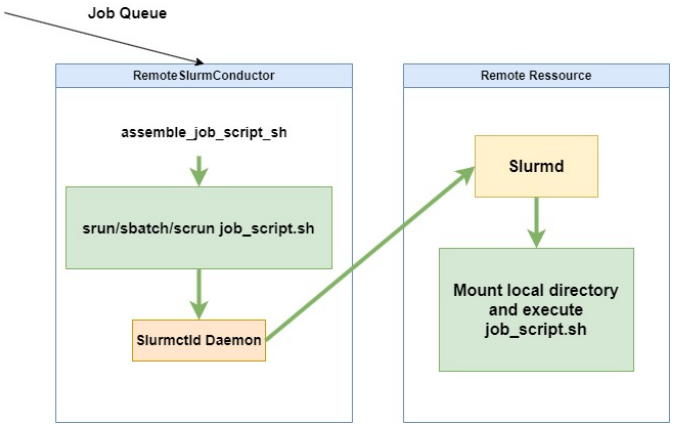

Figure 10: Alternative design where the slurm controller daemon is running on the local machine

metafile particularly often, as the workflow will take many hours to complete anyway. In any case the monitor in MEOW should be used for this purpose. The remote solution should also be able to run both concurrently and in parallel by way of multiple conductors, such that we not only have parallel job processing, but also parallel job scheduling. Multiple threads should also be explored for concurrency. Another thing to add would be some way of enabling the user to upload their own Docker images to a shared repository, instead of having them prebuilt on the remote resource, as this will better enable custom jobs. Furthermore the local sandbox might be hosted somewhere else, such that we don't require the local machine to be active during the entire workflow, but just during the initial scheduling. It will also better enable us to pass the mount point as an argument to the container, thereby removing the need for sys_admin capabilities, while simultaneously making it less important to mount on a per-job basis.
One could also explore a better way of passing arguments to slurm methods, as currently it is slightly janky to have them as entries in a string list. Allowing the user to better specify what goes into connect.sh and startcontainer.sh will help with this, and these files will need additional arguments anyway as the user, host and path to the private key are not arguments to these at the moment.

In the long run the alternative design mentioned previously should be considered and the potential security risk factors associated with it should be evaluated. The current design should also not be considered as security proof, and needs more in-depth analysis to catch potential vulnerabilities. More over it would be interesting to create a more realistic test environment, in the sense of a faster remote machine running multiple compute nodes, and see how the implementation scales. Since slurm is not the only workload manager used one could implement additional conductor instances that adds support for other orchestrators or potentially add support for a generic solution as a middleman as is being explored by the corc project[18]

5 Conclusion

This paper explored how a rules based scheduling system can be integrated with a traditional workload manager such as slurm. Additions to the existing conductor have been made in order to carry out processing on a remote resource. This was done using containers and sand-

boxing both for convenience and security purposes. Tests have been conducted for both directly running the containers and running them with slurm, where both demonstrated some, but linear overhead compared with the local solution. Most of this overhead is caused by connection retries, mount and un-mounting and some for file transfer, however file transfer overhead is expected to increase for a more realistic setup. Additionally tests were done on actual scientific analysis which showed less overhead for sufficient compute nodes. Our results show that no unacceptable overhead has been added, and therefore conclude the integration was a success. Thus we can recommend that this system is used for future work. Attempts were made to make the transfer and processing of these jobs as secure as possible, by limiting the remotes access to the local file system and limiting the privileges of the containers. However more work is needed to make any definitive conclusions about whether or not the attack surface is sufficiently reduced.

References

[1] Spade data movement. http://nest.lbl.gov/projects/spade/html/. Accessed: 2023-April-25.

[2] migrid-sync. https://github.com/ucphhpc/migrid-sync, 2023.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. 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.

[4] Ilkay Altintas, Jessica Block, R.A. Callafon, Daniel Crawl, Charles Cowart, Amarnath Gupta, Mai Nguyen, Hans-Werner Braun, Jurgen Schulze, Michael Gollner, Arnaud Trouve, and Larry Smarr. Towards an integrated cyberinfrastructure for scalable data-driven monitoring, dynamic prediction and resilience of wildfires. *Procedia Computer Science*, 51:1633–1642, 12 2015.

[5] Ryan Chard, Kyle Chard, Jason Alt, Dilworth Y. Parkinson, Steve Tuecke, and Ian Foster. Ripple: Home automation for research data management. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 389–394, 2017.

[6] David Marchant. meow_base. https://github.com/PatchOfScotland/meow_base, 2022.

[7] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. Pegasus: Planning for execution in grids. Technical Report Technical Report 2002-20, GriPhyN, 2002.

[8] Jack Deslippe, Abdelilah Essiari, Simon J. Patton, Taghrid Samak, Craig E. Tull, Alexander Hexemer, Dinesh Kumar, Dilworth Parkinson, and Polite Stewart. Workflow management for real-time analysis of lightsource experiments. In *2014 9th Workshop on Workflows in Support of Large-Scale Science*, pages 31–40, 2014.

[9] Matilda Engström Ericsson. Security in rootless containers: Measuring the attack surface of containers, 2022.

[10] Bas P. Harenslak and Julian Rutger de Ruiter. *Data Pipelines with Apache Airflow.* Manning, 1 edition, 2021.

[11] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[12] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems.* ACM, jul 2020.

[13] David Marchant. Events as a basis for workflow scheduling. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 52–59, 2022.

[14] David Marchant. mig_meow. https://github.com/PatchOfScotland/migmeow, 2022.

[15] David Marchant, Rasmus Munk, Elise O. Brenne, and Brian Vinter. Managing event oriented workflows. In *2020 IEEE/ACM 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop*

Computing (XLOOP), pages 23–28, 2020.

[16] David Marchant Mark Blomqvist. Baproject. https://github.com/Turbodunker/BaProject, 2023.

[17] Marta Mattoso, Jonas Dias, Kary A.C.S.Ocaña, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vitor Silva, and Daniel de Oliviera. Dynamic steering of HPC scientific workflows: A survey. *Future Generation Computer Systems*, 46:100–113, 2015.

[18] Rasmus Munk, David Marchant, and Brian Vinter. Cloud enabling educational platforms with corc. *CTE Workshop Proceedings*, 8:438–457, Mar. 2021.

[19] Diana Adjei Nyarko and Rose Cheuk-wai Fong. Cyber security compliance among remote workers. In Hamid Jahankhani, editor, *Cybersecurity in the Age of Smart Societies*, pages 343–369, Cham, 2023. Springer International Publishing.

[20] Nathan Rini. Docker hearts slurm. In *Presentations from the HPC Containers Advisory Working Group, November 2022*, page 9. SchedMD, 2022. https://slurm.schedmd.com/SC22/Slurm-Hearts-Containers.pdf.

[21] Josephine Wolff. Trends in cybercrime during the covid-19 pandemic. In *Beyond the Pandemic? Exploring the Impact of COVID-19 on Telecommunications and the Internet*, pages 215–227. Emerald Publishing Limited, 2023.