Memory thrifty workflow resource assignment

Bachelor Thesis Exposé

Florian Georg Friederici

August 30, 2023

Abstract

Scientific workflow management systems (SWMS) allow non-tech-savvy persons to express their specific scientific calculation problems in a domain-specific language. To solve them, SWMS get processes executed without requiring in-depth knowledge of the underlying computation systems. The downside of this abstraction is, that the impact of resource requirements from such workflow descriptions on the availability of concrete resources is less tangible. Consequently, it is more difficult for workflow developers to estimate the memory resource requirements. In [10], Lehmann et al. introduce a REST API to exchange workflow scheduling information between SWMS and resource managers (RM). They also presented a scheduler based on this API, which we will extend to facilitate models and algorithms for memory resource estimation to allow a memory thrifty resource assignment. To achieve this, this thesis will research the required background, design an architecture, and build a prototype. We will conduct tests and experiments with the prototype, to validate the approach.

1 Introduction

This thesis will focus on the following scenario: There is a SWMS and a concrete scientific workflow Ψ, that contains many tasks. For example, an Earth observation workflow that analyses hundreds of gigabytes of data from thousands of satellite images, as described in [11].

While different SWMS fulfil the same purpose, their concrete implementation and usage are different. To allow our approach to be generic, we decided to use a model, which is not bound to a specific SWMS implementation. We will use the following notation for the workflow description. We start with an example and explain the elements in the later paragraphs. Figure 1 provides an example of a workflow consisting of four tasks ϑ_1,

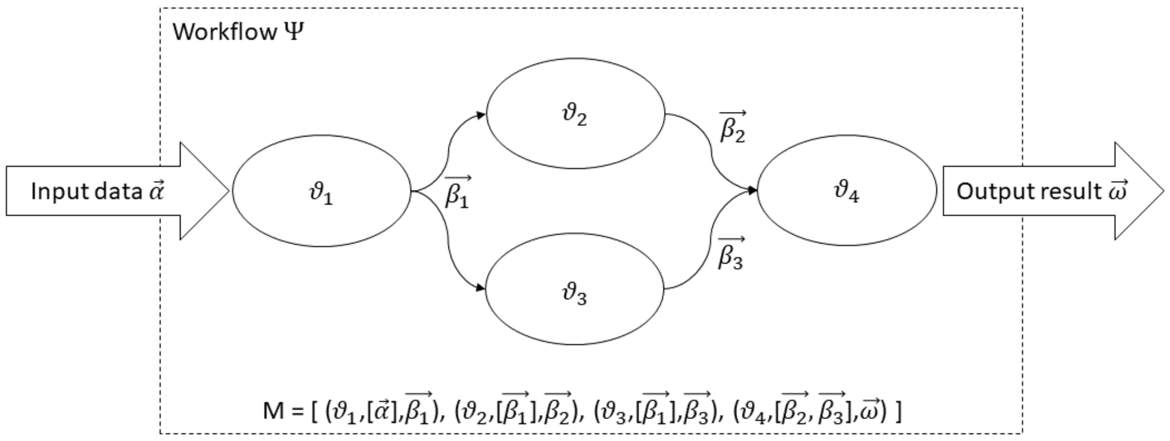
Workflow Ψ
Input data $\vec{\alpha}$
ϑ_1
$\vec{\beta_1}$
ϑ_2
ϑ_3
$\vec{\beta_2}$
$\vec{\beta_3}$
ϑ_4
Output result $\vec{\omega}$
M = [(ϑ_1,[$\vec{\alpha}$],$\vec{\beta_1}$), (ϑ_2,[$\vec{\beta_1}$],$\vec{\beta_2}$), (ϑ_3,[$\vec{\beta_1}$],$\vec{\beta_3}$), (ϑ_4,[$\vec{\beta_2}$, $\vec{\beta_3}$],$\vec{\omega}$)]

Figure 1: Example workflow model with four distinct tasks

ϑ_2, ϑ_3, and ϑ_4. The interconnection, shown by arrows in Figure 1, is stored in a mapping called M.

First, we start with the dataflow on the outer task level: In Figure 1, the first task ϑ_1 takes the input data $\overrightarrow{\alpha}$ and creates some intermediate data $\overrightarrow{\beta_1}$. The intermediate data is the input for ϑ_2 and ϑ_3. These tasks create additional intermediate data $\overrightarrow{\beta_2}$ and $\overrightarrow{\beta_3}$, which is the input to the fourth task ϑ_4 to create the workflow output result $\overrightarrow{\omega}$.

Secondly, Figure 2 provides an insight into the tasks: Every task ϑ_n consists of input $\overrightarrow{i_n}$, output $\overrightarrow{o_n}$, parameters P_n and algorithm A_n. Let's elaborate more about the need for $\overrightarrow{i}$, $\overrightarrow{\sigma}$, and $\overrightarrow{\beta}$. $\overrightarrow{o_n}$ denotes all the output of A_n, regardless of its further use in other contexts. $\overrightarrow{\beta_n}$ is a subset of $\overrightarrow{o_n}$, containing all the data that the workflow will pass on to other tasks. In the example above, $\overrightarrow{i_2}$ is the subset of $\overrightarrow{\beta_1}$, that is the input for the specific task ϑ_2 and $\overrightarrow{i_3}$ is the subset of $\overrightarrow{\beta_1}$, that is the input for the specific task ϑ_3. $\overrightarrow{\beta_2}$ and $\overrightarrow{\beta_3}$ will be joined, and a subset of them will form $\overrightarrow{i_4}$.

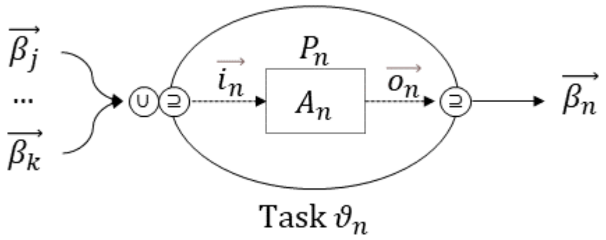

Figure 2: Detail view of a task

In general, a workflow Ψ is the collection of tasks ϑ, the input data $\overrightarrow{\alpha}$, intermediate processing data $\overrightarrow{\beta}$, the processing output result $\overrightarrow{\omega}$, and a mapping M that indicates which tasks are associated to which data input and data outputs. Through the mapping M, at least one task will be associated to $\overrightarrow{\alpha}$, and at least one task will be associated to $\overrightarrow{\omega}$.

Each task ϑ_n is a black box implementation of any algorithmic transformation A_n of input data $\overrightarrow{i_n}$ and parameters P_n to output data $\overrightarrow{o_n}$. For this thesis, we expect the results to be deterministic. We assume that the calculation of A requires a finite time and always finishes, either successful or failing. Tasks ϑ may share the same A and P, and only differ by the data $(\overrightarrow{i}, \overrightarrow{o})$.

At the time of Ψ's start, $(\overrightarrow{\beta}, \overrightarrow{\omega})$ are unknown, and the input data $\overrightarrow{\alpha}$, as well as the tasks ϑ and their mapping M might be incomplete. At the start of each single ϑ_n, we specify that $(A_n, \overrightarrow{i_n}, P_n)$ must be known and $\overrightarrow{o_n}$ is unknown. The SWMS will calculate a directed acyclic graph (DAG) from the known tasks ϑ and their I/O-mapping M.

The makespan of Ψ is the time between the start of the workflow and the end of execution of the latest tasks in the workflow, corresponding to the use of the term in related literature like in [10]. Different factors may impact the makespan of Ψ, such as the amount of processing resources assigned to the tasks, the order and node assignment of the tasks, the complexity and purpose of A, the selection of P, and the size of $\overrightarrow{\alpha}$, $\overrightarrow{\beta}$, and $\overrightarrow{\omega}$.

Witt et al. showed in [18] that a learning approach for memory allocation can surpass user estimates. They further state that "Regression based memory allocation has the highest potential where input sizes and memory usage vary strongly and are highly correlated". Consequently, in this thesis, we will assume that there is a correlation between the size of $\overrightarrow{i}$ and maximum memory usage μ_{max}. We further assume that there is no trade-off between calculation time and memory usage possible, i.e., the algorithms will not dynamically adapt. Even though many real-world tools offer such an option, we decided to exclude this for this thesis.

Every time the resource manager (RM) prepares an execution environment for one ϑ, it must reserve physical resources (i.e., CPU and memory).

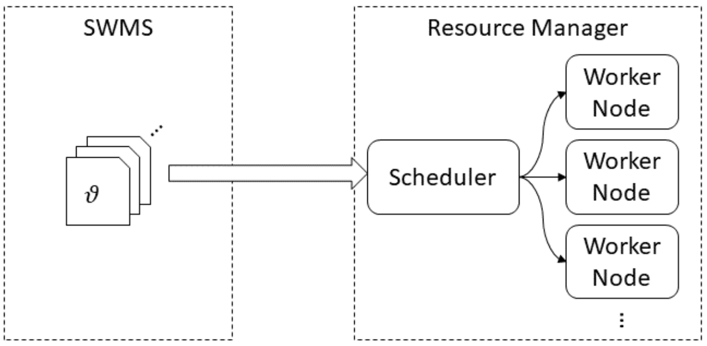

Figure 3: SWMS submits tasks to RM, and they will run on a worker node

The RM will schedule ϑ on a concrete worker node, see Figure 3. But it is important to note that we assume that the RM is not workflow-aware beyond that. Each worker node is subject to CPU and memory limitations. So, the RM must utilize the resources to capacity, to achieve a short makespan.

Real-world scenarios often use Kubernetes as a RM. The SWMS will wrap every ϑ

into a container, and those into Kubernetes pods. For memory constraints, there are two parameters that can be specified: ‘requests’ and ‘limits’. Those values do have a massive impact on the scheduling and execution of pods. According to the documentation in [1], ‘request’ formulates a lower, and ‘limits’ an upper bound on the memory available. The pod only will get scheduled when the RM can grant the lower value. The RM will terminate the pod if it tries to exceed the limit.

In today’s realizations, the workflow developers must provide the values for resource requirements if they would rather not rely on inappropriate, static defaults. For example, using Nextflow as a SWMS, there are the ‘cpus’ and ‘memory’ directives. Those can be static values, or dynamically increase, as described in [13]. If the memory estimation for a ϑ is too generous, resources are left unused. On the other hand, if too little memory is assigned, ϑ will terminate with an out of memory error and ϑ must be rerun, which might increase the makespan. In [16], Tovar et al. discussed this job sizing problem.

2 Research Question

Witt et al. made comparisons of predictive performance modelling research approaches in [17]. Scientific workflows with a huge number of similar tasks present an opportunity to test and validate such models against real-world workflows and data. Prior work by Witt et al. in [18] was based on captured data. This thesis will focus on the integration of models and algorithms for memory resource estimation to allow a memory thrifty resource assignment during SWMS and RM runtime. We will investigate the following questions:

1. Which models or algorithms are suitable for a realistic prediction of memory requirements for our scenarios, without the presence of historical data?
2. How to integrate a module that can learn and estimate the memory requirements of a task into a system of SWMS and RM, where to interface with the existing components?
3. How to design this module in a way, that can incorporate different algorithms or models?
4. How fast can the estimation converge to values that do impact the makespan?
5. How does the scheduling affect the learning progress?

We will do the practical evaluation as an extension to the ‘Common Workflow Scheduler for Kubernetes’ (CWS) [9]. The expected outcome of this work will include the theoretical foundation, experiment setups and measurements, which we will document in the thesis, as well as the concrete implementation.

3 Related Work

There are many scientific workflow management systems (SWMS) available. Examples of SWMS are: Nextflow [8], Snakemake [6], and Pegasus [20]. The main task of SWMS is to automate the execution of data analytics workflows. The SWMS therefore offers an abstraction to separate the workflow's logic from the concrete execution environment. Nextflow, offers a Groovy-based domain-specific language (DSL) for the workflow description, and can then execute it on multiple platforms. Snakemake uses a 'Snakefile' with rules that specify the workflow in small steps. Pegasus uses a declarative YAML file to describe the workflow and converts it into a directed acyclic graph (DAG).

A SWMS typically sends the workflow tasks to a RM, which is responsible for executing the concrete data analytics workflow steps, cf. Figure 3. The RM manages the compute resources in the cluster, and is responsible for orchestration, scaling, and management of applications. This work will focus on Kubernetes [2] as a RM. Among the advantages of Kubernetes are, the wide adoption across the industry and the capability to be either self-hosted or rented as a managed service from cloud providers. Kubernetes can even run locally on a single device for development purposes, for example with minikube [3]. Besides Kubernetes, other resource managers like HTCondor [12] or Slurm [15] exist.

In [10], Lehmann et al. introduce an interface that allows the exchange of workflow related information between the SWMS and the RM. Their proof-of-concept (PoC) implementation is the 'Common Workflow Scheduler for Kubernetes' (CWS) [9]. The available PoC implementation uses Nextflow and Kubernetes. Furthermore, Lehmann et al. discuss the impact of different workflow scheduling algorithms and the scheduling of SWMS tasks on resource managers.

In [5], Bader et al. address the resource allocation issues that arise in heterogenous clusters. They claim that beyond the rough metrics, like core count or total memory, fine-grained heterogeneity aspects do make performance differences, that impact the prediction of tasks duration and therefore lead to suboptimal allocation choices. They also base their work on Nextflow and Kubernetes, and their approach to task monitoring and labelling is of interest for our proposed work.

In [18], Witt et al. suggests the 'low-wastage regression' (LWR) method to reduce the amount of wasted memory and compares it with the approach from Tovar et al. [16]. In [19], Witt et al. present an approach to learn peak memory usage of tasks. Both [18] and [19] use the 'Memory Allocation Quality' (MAQ) as a measure to characterize the quality of a scheduler's memory allocation decisions.

4 Approach and Methodology

To address the research questions, we will break the problem down into three sub-problems and solve them individually. First, how to get statistics and monitoring data from the running tasks. Secondly, how to construct a model for memory usage prediction. And thirdly, how to feed back the prediction data and change future tasks accordingly. We will discuss those three sub-problems in the following paragraphs. Figure 4 illustrates the

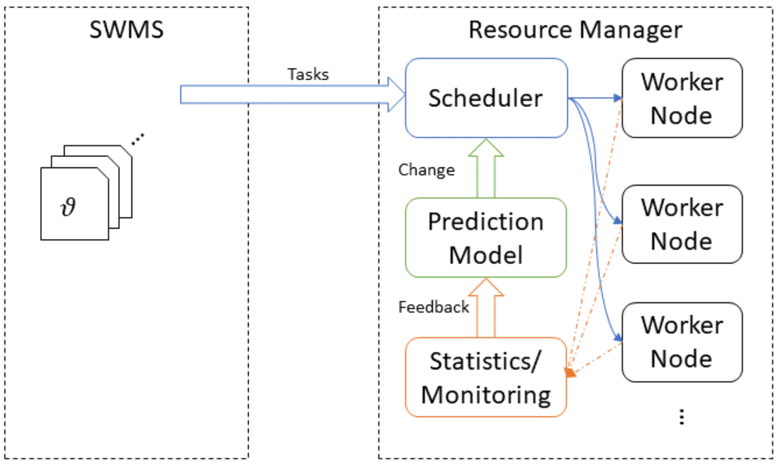

Figure 4: Additions to the systems architecture

systems architecture and where the building blocks for those sub-problems are located.

Statistics/monitoring of tasks ϑ: To provide the second sub-problem with the necessary input data, we need to collect data about the size of $\overrightarrow{i}$ for each task ϑ and monitor the maximal memory usage μ_{max} during execution. There are two main possibilities to gather such data: The Nextflow integrated tracing [14], or the Kubernetes integrated metrics [4].

Model for memory prediction: As per definition (A, $\overrightarrow{i}$,P) will be known on the start of each ϑ. After the execution of ϑ, $\overrightarrow{o}$ will be known as well, and additionally μ_{max} from the statistics/monitoring information subtask. The purpose of this second sub-problem will be to derive a proper model from the observed relationship between (A, $\overrightarrow{i}$,P) and μ_{max}, so that we can estimate μ_{max} from (A, $\overrightarrow{i}$,P) of future ϑ. This also includes determining which combination of (A, $\overrightarrow{i}$,P) observations are sufficiently similar to each other to be meaningful.

Feedback of the prediction data: In the current implementation, Nextflow sends all tasks to Kubernetes right after the start of the workflow. We will use the workflow-aware Common Workflow Scheduler [9] and extend it so that it changes the task's memory requirements from our predictions before the tasks start. Kubernetes pods resource specifications were immutable in the past, so any change would require the task to be resubmitted. However, thanks to KEP-1287 [7], this will change in upcoming Kubernetes versions.

The concrete experimental setup will include two variations. One, that we can easily start on a single machine for development and troubleshooting purposes. And a second that mimics a real-world environment with distributed worker nodes on physical or virtual hardware.

5 Goals and Limitations

We will set up a SWMS with RM and integrate our prototype implementation of memory prediction into it. The technological foundation will be oriented on the works of [10] and [5], i.e., it will be based on Nextflow and Kubernetes and aims to extend the Common Workflow Scheduler [9]. This will be the basis for our experimental evaluation and be available for the public under a permissive open-source licence to replicate our results, as well as extend it for future research. Our experiments will target the following points:

- Effective interfacing with existing components
- Modular construction for replaceable models or algorithms
- Fast prediction model convergence
- Effects of scheduling to the learning progress

We expect that our approach cannot derive a memory prediction model under the following circumstances:

1. Low number of ϑ. The threshold is subject to this work.
2. Very different ϑ. We will evaluate the impact of the characteristics.
3. Suitable ϑ are already running. Even if 1 and 2 are met, the approach can only improve the execution of future tasks.

Given those limitations, we will focus the evaluation on synthetic workflow descriptions with specially crafted tasks that target our research questions. We will consult real-world workflows to ensure general applicability whenever feasible.

References

[1] The Kubernetes Authors. *Assign Memory Resources to Containers and Pods.* URL: `https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/` (visited on Aug. 13, 2023).

[2] The Kubernetes Authors. *Kubernetes.* URL: `https://kubernetes.io/` (visited on Aug. 13, 2023).

[3] The Kubernetes Authors. *Welcome! — minikube.* URL: `https://minikube.sigs.k8s.io/docs/` (visited on Aug. 13, 2023).

[4] The Kubernetes Autors. *Node metrics data — Kubernetes.* URL: `https://kubernetes.io/docs/reference/instrumentation/node-metrics/` (visited on Aug. 13, 2023).

[5] Jonathan Bader, Lauritz Thamsen, Svetlana Kulagina, Jonathan Will, Henning Meyerhenke, et al. "Tarema: Adaptive resource allocation for scalable scientific workflows in heterogeneous clusters". In: *2021 IEEE International Conference on Big Data (Big Data).* IEEE. 2021, pp. 65–75.

[6] Johannes Köster. *Snakemake - A framework for reproducible data analysis.* URL: `https://snakemake.github.io/` (visited on Aug. 13, 2023).

[7] Vinay Kulkarni and contributors. *In-Place Update of Pod Resources #1287*. URL: `https://github.com/kubernetes/enhancements/issues/1287` (visited on Aug. 25, 2023).

[8] Seqera Labs. *A DSL for parallel and scalable computational pipelines — Nextflow*. URL: `https://www.nextflow.io/` (visited on Aug. 13, 2023).

[9] Fabian Lehmann, Jonathan Bader, Friedrich Tschirpke, Lauritz Thamsen, and Ulf Leser. *Common Workflow Scheduler for Kubernetes*. Version v1.0. May 2023. DOI: `10.5281/zenodo.7603176`. URL: `https://doi.org/10.5281/zenodo.7603176`.

[10] Fabian Lehmann, Jonathan Bader, Friedrich Tschirpke, Lauritz Thamsen, and Ulf Leser. *How Workflow Engines Should Talk to Resource Managers: A Proposal for a Common Workflow Scheduling Interface*. 2023. arXiv: `2302.07652 [cs.DC]`.

[11] Fabian Lehmann, David Frantz, Sören Becker, Ulf Leser, and Patrick Hostert. "FORCE on Nextflow: Scalable Analysis of Earth Observation Data on Commodity Clusters." In: *CIKM Workshops*. 2021.

[12] HTCondor team at UW-Madison. *Home*. URL: `https://htcondor.org/` (visited on Aug. 13, 2023).

[13] S.L. Seqera Labs. *Processes — Nextflow 23.04.1 documentation*. URL: `https://www.nextflow.io/docs/latest/process.html#dynamic-computing-resources` (visited on Aug. 13, 2023).

[14] S.L. Seqera Labs. *Tracing & visualisation — Nextflow 23.04.1 documentation*. URL: `https://www.nextflow.io/docs/latest/tracing.html` (visited on Aug. 13, 2023).

[15] Slurm Team. *Slurm Workload Manager*. URL: `https://slurm.schedmd.com/overview.html` (visited on Aug. 13, 2023).

[16] Benjamin Tovar, Rafael Ferreira da Silva, Gideon Juve, Ewa Deelman, William Allcock, et al. "A job sizing strategy for high-throughput scientific workflows". In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2017), pp. 240–253.

[17] Carl Witt, Marc Bux, Wladislaw Gusew, and Ulf Leser. "Predictive performance modeling for distributed batch processing using black box monitoring and machine learning". In: *Information Systems* 82 (2019), pp. 33–52.

[18] Carl Witt, Jakob van Santen, and Ulf Leser. "Learning low-wastage memory allocations for scientific workflows at icecube". In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 233–240.

[19] Carl Witt, Dennis Wagner, and Ulf Leser. "Feedback-based resource allocation for batch scheduling of scientific workflows". In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 761–768.

[20] Pegasus WMS. *Pegasus WMS – Automate, recover, and debug scientific computations*. URL: `https://pegasus.isi.edu/` (visited on Aug. 13, 2023).