Bachelorarbeit im Rahmen des Studiengangs
Scientific Programming

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 – Medizintechnik und Technomathematik

A concept study of a flexible asynchronous scheduler for dynamic Earth System Model components on heterogeneous HPC systems

Jülich, November 16, 2020

Jan Vogelsang

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit mit dem Thema

A concept study of a flexible asynchronous scheduler for dynamic Earth System Model components on heterogeneous HPC systems

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Name: Jan Vogelsang

Jülich, den 13.10.2020

J. Vogelsang

(Unterschrift der Studentin/ des Studenten)

Diese Arbeit wurde betreut von:

1.Prüfer: Prof. Dr.-Ing. U. Stegelmann

2.Prüfer: PD Dr. Martin Schultz

Climate change is presenting one of the biggest challenges for mankind as recent developments have shown. Since 1990, the global temperature increased by almost 1°C [1] and earth system model projections indicate that temperatures will raise another two to four degrees by 2100 unless drastic measures are taken quickly to avoid greenhouse gas emissions. Since several decades, scientists have constructed numerical models to simulate weather and climate. Such models describe physical and biogeochemical processes in the atmosphere, the ocean, the land surface and the cryosphere and are thus called earth system models (ESM).
In that same period of time the computing power of the worlds largest supercomputers has rocketed upwards as today's fastest supercomputer offers 170.000 times more computing power than the fastest one 20 years ago [2, 3]. By utilizing this enormous computing power, it has become possible to simulate high-resolution weather and climate models that are capable of predicting extreme events with reasonable accuracy.
As the computing power of modern supercomputers increases rapidly, so does the complexity of the underlying architecture. Specialized nodes equipped with new technology like graphical processing units allow a massive reduction of computing time for many problem classes, but do also introduce the challenge to work with a heterogeneous architecture. ESMs were hitherto designed for homogeneous architectures based central processing units. Along with the increased computational demands of ESMs, the amount of generated data does as well, leading to the phenomenon that the cost of data movement starts to dominate the overall cost of computation. Any new programming paradigm for ESMs must therefore try to minimize massive data transfers, e.g. by utilizing data locality as it will be demonstrated in this work.
Facing the challenges of modern supercomputer architecture and the need for more flexible and modular models, completely new programming concepts are needed, as demonstrated by the Helmholtz project *Pilot Lab Exascale Earth System Modelling* (PL-ExaESM) [4] in which context this work has been conducted.
As a potential solution for these challenges a new, asynchronous scheduling method for modular ESM components has been tested in a sandbox environment and evaluated with respects to performance, scalability and flexibility. Asynchronous scheduling allows for a better exploitation of the heterogeneous resources of a modern HPC system. Through careful consideration of data flow paths across the coupled pseudo-ESM components, data movement could be reduced by more than 50% compared to a traditional sequential ESM workflow.
Furthermore, running different example workflows showed a high efficiency gain for complex workflows when increasing the number of nodes used for computation.
The results obtained here are promising, however not yet sufficient to propose asynchronous scheduling as the one new ESM paradigm to be used for upcoming exascale earth system modelling. Further development and investigation following the approach proposed in this work is required to evaluate the usability on different architectures and comparing it to different approaches meeting the introduced challenges of modern ESM development.

Nomenclature

API	Application Programming Interface
CESM	Community Earth System Model
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
ESM	Earth System Model
GPU	Graphics Processing Unit
gRPC	gRPC Remote Procedure Calls
HPC	High Performance Computing
IO	Input/Output
JUWELS	Jülich Wizard for European Leadership Science
MPI	Message Passing Interface
NetCDF	Network Common Data Form
OpenMP	Open Multi-Processing
RPC	Remote Procedure Call
Slurm	Simple Linux Utility for Resource Management/Slurm Workload Manager
TPU	Tensor Processing Unit

Contents

1 Introduction

1.1 Problem statement

In recent years the computing power of modern supercomputers increased to a point where it enables development of climate and weather simulations with high temporal and spatial resolution which are now also capable of predicting extreme events with reasonable accuracy. The goal for future *earth system model* (ESM) development will now be to achieve a global spatial resolution of 1 kilometer, which would allow explicit calculation of physical equations to reduce the amount of necessary parametrizations. This would however require an increase in processing power ESMs are utilizing of about 1000 times. To allow ESMs to use such an enormous computing power, all workflows have to be perfectly optimized for the hardware powering the computation. Modern ESMs will not only have to adapt to changing architectures caused by reaching the physical limitations regarding the size of transistors, but also handle enormous amounts of data that will be generated. Furthermore, state-of-the-art technologies like machine-learning tend to augment or even replace numerical solvers of differential equations introducing a new challenge for traditional monolithic ESMs. Therefore, a novel approach to modern earth system modeling will be examined in this thesis.

1.2 Earth System Modeling workflows

1.2.1 What are Earth System Models?

The *Southern Ocean Carbon and Climate Observations and Modeling* (SOCCOM) defines ESMs as follows [5]:

> *A coupled climate model is a computer code that estimates the solution to differential equations of fluid motion and thermodynamics to obtain time and space dependent values for temperature, winds and currents, moisture and/or salinity and pressure in the atmosphere and ocean. Components of a climate model simulate the atmosphere, the ocean, sea, ice, the land surface and the vegetation on land and the biogeochemistry of the ocean.*

Earth System Modeling describes the assembly of earth system models as well as the creation of workflows needed in order to read, exchange and write data which is used and generated by these models.

When assembling multiple models, each model will be called component in the resulting coupled model.

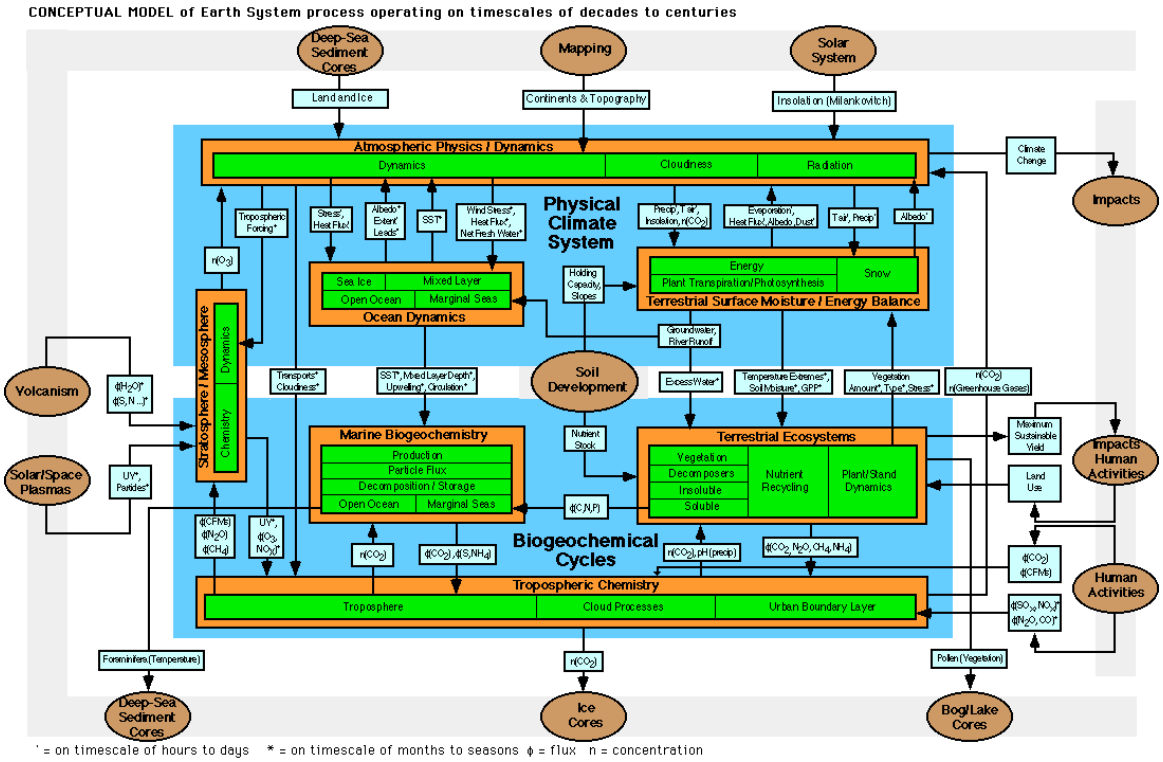

Figure 1.1: An Earth-System-Model consists of multiple interconnected components that exchange information to model a complete earth system. [6, p. 24]

Due to the complex interactions between various earth system components, an ESM consists of many different modules, which interact with each other. As seen in figure 1.1, the different components add up to a workflow consisting of directional dependencies between the components, which effectively leads to extensive exchange of generated data.
As these workflows do therefore require up to terabytes of disk space and many gigabytes of concurrent memory and run computationally intensive simulations, supercomputers or large clusters are necessary to run the simulations in a reasonable time. Execution time is very important for ESM workflows as results will only be meaningful when every component generates and receives data regularly. Extensive communication between coupled compartments and spatial regions in the ESM leads to the necessity of a remarkably high bandwidth to exchange massive amounts of data without excessively restraining computation. Such a high bandwidth between computation units can presently only be guaranteed by supercomputing systems that, in contrast to cloud solutions, interconnect nodes directly for high-speed communication.

Traditional coupled models, like the *Community Earth System Model* (CESM) [7], are built for a specific architecture which makes it very difficult to adapt them to new computer systems and oftentimes not dynamic when it comes to the size of the allocation on a supercomputing system.
Nowadays, most supercomputers offer nodes equipped with *graphics processing units* (GPUs) which are able to run some simulations significantly faster than *central processing units* (CPUs) that are normally used for computation. However, the number of these special nodes is limited on most systems which leads to the special case where one compute job consists of multiple types of nodes. Legacy models are mostly not able to exploit these changing, or in general heterogeneous, architectures.
Using tools like the *Common Infrastructure for Modeling the Earth* (CIME) [8] monolithic models are created that strongly couple multiple components together to a static single executable with an enormous code size. This does usually result in an application that trades flexibility for performance and usually runs very efficiently on the specific architecture it was built for. Once built however, the new model forms a blackbox that can not be modified anymore without rebuilding the whole model. This makes the model very hard to extend and does not allow any dynamic changes during runtime.
Altogether the result is a long developmental period due to the fact that typical trial and error development approaches lead to many completely new builds of the coupled model even when only minimal changes were made to the component structure or the workflow. Additionally, the development architecture has to match the one used in production which may lead to unpredictable results when porting the model to another machine as the scale of the model is predefined and static and can therefore not be scaled up on the production architecture without rebuilding the model.

1.2.2 Concept study: Future ESM workflows

Considering the drawbacks of static, monolithic ESMs, it will be interesting to investigate if the new generation of ESM workflows can become more dynamic and flexible on every architecture to therefore decrease development times.
Instead of coupling components directly (*Peer-to-Peer*), using the *Hub-and-Spoke* paradigm [9] where multiple components (spokes) are connected to one central process (hub), components could be coupled weakly to allow dynamic addition and removal of any component, even at runtime.
Each component would then use its own workflow that is isolated from all other components. This includes both reading from and writing data on a node as well as the actual computation in between. This forms a self-contained application for each component that can be built separately from any other component. By not building the coupled model with all components at once, the whole workflow would get modular and dynamic. Dataflow and execution of components have to be managed at runtime and can therefore not run synchronously in a predefined order anymore but requires asynchronous data and task management. Asynchronous data management

and task scheduling allows dynamic changes of components and a high scalability. Asynchronous scheduling will be covered in-depth in section 1.4.
By designing the ESM workflow this way, it is possible to adjust the number of resources used for a run of the simulation at the moment when submitting the job or optionally even during runtime without having to rebuild the model. This reduces development times significantly as the workflow can be easily tested using smaller node allocations compared to the one that will be used in production. Apart from faster iteration it does also enable running the workflow on changing and even heterogeneous architectures. As each component will be self-contained and isolated, it is possible to run particular components on specific nodes that are, for example, equipped with GPUs or larger memory capacity.
Furthermore, the resulting flexibility of using loosely coupled components opens up entirely novel possibilities of earth system modeling, such as dynamic insertion of ensemble-simulations, on-demand coupling of model components, online-visualization, etc. all at runtime.

1.3 Supercomputer architecture

The first computer ever considered supercomputer, called *Control Data Corporation* (CDC) 6600, reached a peak performance of 3 million *floating point operations per second* (flops) in 1964 [10]. Since then the computing power of supercomputers steadily increased as the number of transistors in CPUs doubled about every two years according to Moore's law [11]. This trend has been ongoing for about 50 years now, however, today transistors can only hardly be shrunk any further and people start claiming the end of Moore's law [12].
As a consequence, modern supercomputers start to introduce different ways of increasing computational power and improving the overall architecture. Accelerators are being used next to traditional CPUs such as GPUs leading to incredible peak performances of up to 415.5 petaflops [3].
To develop new solutions for ESMs, these and other new components, such as highly efficient storage systems, of a supercomputer must be analyzed and their characteristic properties must be known in order to write efficient code.

1.3.1 JUWELS supercomputer

This work was performed in the *Jülich Supercomputing Centre* (JSC) utilizing the *Jülich Wizard for European Leadership Science* (JUWELS) supercomputer [13].
The supercomputer provides a cluster of multiple different nodes, ranging from standard compute nodes to visualization nodes with up to 768 GB of main memory, two terabytes of on-board disk space and high-performance Nvidia Pascal P100 GPUs. The standard compute nodes are equipped with 2 Dual Intel Xeon Platinum 8168

CPUs each with 24 cores at a clock rate of 2.7 GHz and up to 96 threads per node using Intel Hyperthreading technology as well as 96 GB of 2666 MHz main memory. The system is being operated by the CentOS 7 Linux distribution.
All nodes are connected using Infiniband technology allowing high-bandwidth inter-node communication with a throughput of up to 100 GB/s. The supercomputer is attached to the *Jülich Storage Cluster* (JUST) which serves as central storage provider using a parallel file system from IBM, called *Spectrum Scale*, which is used on the system to provide a centralized storage method that can be accessed from all nodes. It offers a capacity of 52 petabytes and a peak bandwidth of 380 GB/s citejust.

1.3.2 Heterogeneous architecture

Many modern supercomputers, such as the JUWELS supercomputer, begin to offer a variety of different types of nodes having varying numbers of CPUs, being equipped with GPUs, having an increased memory capacity or even providing on-node disk space. However, there are often far less of these specialized nodes available on the system. Therefore, if a workflow wants to make use of these specialized nodes to potentially reduce computation times of specific components, it is required to combine different node types in a single node allocation. Using these heterogeneous architectures efficiently in an application is a very challenging task though.
The central hub-process now also has to manage all available nodes and assign components to specific nodes explicitly while keeping track of utilization of each node.

1.3.3 Data intensive tasks

ESM components can be very complex, but the code itself is often highly optimized to compute results as fast and efficiently as possible. However, especially when combining multiple components, they are not optimized for the architecture they will run on. Most traditional ESMs simply combine existing efficient components without specifically optimizing the resulting coupled workflow which leads to poor efficiency on specific architectures.
The main bottleneck of coupled ESMs is usually presented by long IO times as up to terabytes of data will be exchanged between the different components in a single run of the workflow which often leads to the cost of data movement dominating the overall cost of computation.
As by today big data became a relevant matter in most fields of research, the supercomputer architecture gets adapted to these novel demands and offers both hard- and software solutions to optimize such data-intensive workflows.

1.3.3.1 Hierarchical storage architecture

Many supercomputer architectures offer smaller storage systems with less latency and often also higher bandwidth compared to the ones with larger capacity. On the JUWELS supercomputer there is a shared *High-Performance Storage* (HPST) next to the standard shared scratch storage. The HPST system provides less latency by caching frequently used data in a fast low-capacity storage to reduce access times.
Some supercomputing systems also provide actual disk space on each node next to the node's memory. Temporarily storing data on nodes where the data will be used repeatedly can significantly reduce IO times.

1.3.3.2 On-node memory

On many supercomputers there are no on-node disks, however each node will always have its own memory. On Unix-like operating systems it is possible to actively store data on nodes by utilizing temporary in-memory filesystems like *tmpfs*. This allows reading and writing data from a filesystem which is required by nearly every ESM component, while preserving the unbeatable performance that volatile *random access memory* (RAM) provides.
Mounting filesystems on these systems typically requires root privileges which most supercomputer users will not own. A solution to this problem is provided by the in-memory temporary filesystem that is mounted at `/dev/shm` on most Unix-like operating systems, such as the *CentOS* that powers the JUWELS supercomputer. This mount enables easy access to each node's memory and lets the workflow read and write files just like it would on any other filesystem.

1.3.3.3 Parallel filesystems

Assisting with the need for flexible, high-performance filesystems, many solutions for parallel multi-node filesystems have been created, the most promising one on *high-performance computing* (HPC) systems probably being *BeeGFS* [14]. BeeGFS is a parallel cluster filesystem that essentially combines several filesystems of nearly any type to a single combined parallel cluster filesystem. However, as BeeGFS was not available on the JUWELS supercomputer at the time this work was performed, it has not been used or further discussed in this thesis.

1.4 Synchronous vs asynchronous scheduling

In this work instead of using an existing job scheduler like *Slurm* [15] to manage the execution of the different components, a new scheduler has been developed to have more control over the scheduling behavior. To avoid Slurm scheduling the different tasks, a job allocation with a sufficient number of compute nodes will be allocated

for the whole workflow in advance. The scheduler then uses the assigned nodes of this allocation itself by scheduling tasks efficiently avoiding idling of as many CPUs as possible. This step has been necessary for multiple reasons. First of all, when submitting a job to slurm, is is unclear at which point in time this job will be scheduled and therefore when the task will run. For tasks depending on other tasks, this can easily result in highly inefficient scheduling compared to self-controlled scheduling. Furthermore, when submitting a job, random nodes will be allocated which makes it impossible to benefit from data locality by storing data on on-node memory.
The newly created scheduler works asynchronously and schedules tasks based on events. Synchronous schedulers on the other hand are best suited for periodic tasks with job-wide synchronism by using a common clock. While periodicity definitely exists in ESM workflows, asynchronism adds the option to react differently to unforeseen events, e.g. a hurricane, by deviating from a fixed, synchronous schedule. The additional flexibility makes it easier to prioritize further analysis of these events over other routine calculations by, for example, running an ensemble of tasks in that specific region where the event occured. Such events are typically analyzed using ensembles that run simulations for the same, fixed region either using slightly varying input parameters or generate data for a different time period.

1.4.1 Comparison of scheduling approaches

Asynchronous scheduling is a well-established software paradigm, which is however rarely applied in ESM codes, largely due to legacy reasons. It promises several advantages for future ESM workflows including better average case performance, more accurate runtime estimation and most importantly significantly more flexibility.
As the time a component takes to run one iteration is a priori unknown and can vary a lot depending on the resources it was assigned, it would be difficult or even impossible to schedule the tasks efficiently in a synchronous way which would imply setting a fixed order and timing for all tasks. If specific tasks would always wait for other tasks, long idle times would occur which in the worst case leads to very poor performance. In the best case, however, synchronous scheduling should perform better compared to asynchronous scheduling, as the latter always involves some overhead due to additional scheduling logic at runtime instead of scheduling all tasks before the actual execution.
Using asynchronous scheduling, idle times can be reduced to almost zero as tasks can be scheduled freely and do not have to follow each other directly. Therefore tasks requiring few CPUs can be used to fill holes while larger tasks would be prioritized to let those run if possible and only use smaller tasks otherwise. In both scheduling scenarios however, a task depending on the results of another task will have to wait for it to finish before it can run. The timing of task executions will therefore always depend on inter-process dependencies.
Efficient data management presents a key factor for high-performance asynchronous schedulers. As tasks can in theory run on an arbitrary node, the scheduler will try to

run tasks on the same node where their input data is located. This can significantly reduce IO times compared to synchronous schedulers that cannot make use of data locality.
When scheduling asynchronously, the efficiency can be further improved at runtime as the computing time of one iteration of a component can be measured once and will then be estimated relatively accurate for all following iterations. While synchronous schedulers can schedule tasks perfectly without any overhead when the computing time of every component is known beforehand. If the computing time is unknown or changes dynamically due to a task getting assigned less or more resources, the scheduler cannot accurately estimate the computing time anymore. Therefore, it will produce less efficient results compared to an asynchronous scheduler even in the average case.
A significant advantage of asynchronous scheduling is its capability to use any architecture to its full extent at any time. In case of heterogeneous architectures, the tasks can be scheduled to the nodes they run most efficient on. If, for example, a node equipped with a GPU becomes available, the scheduler can dynamically at runtime decide to run a specific task on that node instead of a simple node consisting of CPUs only.
As the scheduler can easily adapt to new tasks that are waiting to get included in the schedule, completely new tasks can be added during runtime and could even get prioritized over other tasks. Tasks can also represent any other form of component and use the generated data for online data visualization, post-processing of results, or as inputs to a finer-resolved regional simulation, for instance.
As tasks can be added during runtime, they can of course also be removed or replaced. This makes components interchangeable, one could, for example, swap a numerical model component by a machine learning model or replace a low resolution model with a high resolution one for a specific region. This will be especially important for the simulation of extreme events (e.g. heavy precipitation) that require very high spatial and temporal resolutions.

1.5 Study concept

The goal of this thesis is to evaluate the possible advantages of using an asynchronous scheduling approach for future ESM workflow development compared to traditional synchronous scheduling.
Therefore, the concept for an asynchronous scheduler had been designed from the ground up and afterwards implemented using the Python programming language. For subsequent benchmarking, a sandbox ESM workflow had been created and implemented which already profited from the several optimizations that were made in conjunction with the asynchronous scheduler. Finally, some defined benchmarks had been evaluated, namely memory efficiency, performance compared to a synchronous version, scalability and flexibility.

2 Methods

2.1 Sandbox asynchronous ESM workflow

As discussed in the previous chapter, ESM workflows are enormously complex and effectively evaluating the functionality, as well as benchmarking whether the desired results can be achieved using this new approach for ESM, is nearly impossible when starting with a complete ESM workflow. Instead, a sandbox workflow will be created, simulating the runtime and IO behavior of typical ESM components.
To accurately simulate ESM workflows in their entirety, all possible components a workflow may consist of have to be substituted by here called tasks emulating their behavior and runtime characteristics.
All tasks as well as the scheduler itself are implemented using the Python programming language [**source-code**]. Keep in mind that typically ESM components are written in either C++ or Fortran for increased performance, however python scripts and compiled Fortran or C++ scripts can be used interchangeably as both can be directly invoked from a shell by using python's *subprocess* module.
To be concrete, these tasks have to emulate the following functionality.

2.1.1 Functionality

2.1.1.1 Reading data

Almost every component in an ESM workflow will read in some data at some point during its lifetime. Reading in data is as simple as opening the file, as all data management will be handled by the Scheduler itself. More information about data handling can be found in section 2.2.3.
In this sandbox scenario, python's built-in `open` routine in combination with the `'rb'` binary read option will be used to read in data from a file.

2.1.1.2 Writing data

Most components will generate data, usually after processing some input data. However, not all components will actually write data, a visualization component, for example, will probably not write any output.
In this sandbox scenario, python's built-in `open` routine in combination with the `'wb'`

binary write option will be used to write data into a file.

2.1.1.3 Sending and receiving data

Due to the asynchronism of task execution, no direct communication between tasks is possible, as that would require them to run synchronously for the time of communication. As asynchronous tasks do not share the same clock, synchronous execution can not be achieved.
However, tasks can communicate indirectly using data they produce and consume. The *producer and consumer* design pattern is a commonly used pattern when working with tasks that run independently from each other. The task that wants to send data, the producer, writes its data into a file without knowing which other task, the consumer, will receive the data by simply reading from the file.
By implementing so called data brokers, data will actively be conveyed between the tasks. They allow tasks to inform other tasks about their newly generated data without having to directly communicate with them.
Brokers could in theory even influence the scheduling of other tasks, for example by asking the central scheduling task to increase priority of a data consumer task. More information about data brokers will be provided in section 2.2.1.

2.1.1.4 Processing data

Between reading and writing data, most tasks will process the input to generate their output data.
That process can be classified to two sub-classes, exact and iterative algorithms. Some mathematical problems can be solved exactly with diverse algorithms that will take a constant amount of time.
To simulate these kind of algorithms, python's `sleep` routine of the `time` module is used with a constant time value representing the time it takes for the complete calculation to process.
Many problems cannot be solved directly and therefore require implicit algorithms that find an approximate solution for the problem. Most of these algorithms will run multiple iterations, stopping the calculation as soon as some criteria is met. The number of iterations needed is not known beforehand resulting in an inconstant amount of time being needed for the calculation.
To simulate these kind of algorithms, again the `sleep` routine is used. This time, the amount of time the process will sleep is not a constant value, but instead a multiplication of a fixed value representing the time one iteration takes to process and a randomly generated number within a reasonable range representing the number of iterations being run.

2.1.2 Scheduler and model time

Understanding the difference between scheduler and model time in an asynchronous scheduling scenario can be quite confusing at first. One would probably expect both to be the same in the sense that the model time runs concurrent to the scheduler time. That would mean at any given point in time one could get an exact value for the model time given the scheduler time and vice versa. However, as there is no shared clock between the different tasks the time step a task is currently simulating does not have to match the time step of other tasks. And not only the time step can vary but even the whole time interval can be different. This can happen when a task is getting scheduled which is ahead of other tasks in time. As the dependencies between tasks usually are non-linear, it will happen that some tasks currently do not have to wait for any input data and can therefore already simulate the next time interval. The difference in model time between the tasks will usually not vary too much though, as sooner or later a task will have to wait for input from another task again.

2.1.3 Workflow

The sandbox workflow is run on the JUWELS supercomputer only using compute nodes within the scope of this work. However, the ability to use any compute node without any prior configuration already implies that the scheduler is designed in a flexible way, therefore not being limited to special nodes or a homogeneous node configuration.

To cover all functionality of ESM components, four different tasks are needed which are combined to a single workflow as shown in figure 2.1. The initial task, called Read/Write component, starts with a dataset of about 300 megabytes in size.
In reality, most components will run continuously instead of only running once. For testing purposes however, all components are only run once.

Most ESM components usually parallelize the computation by using multiple cores on one node or running multiple nodes concurrently. To simulate this behavior, all tasks except the data broker tasks will be provided multiple cores and partly also multiple nodes.
Typically, ESM components combine most of the above functionality at once, e.g. read data, process that data, write data to a file and finally inform other components about the generated data.
However, for testing purposes the functionality will be isolated as much as possible as it does allow for more accurate benchmarking and evaluation. In the following chapters, this workflow will be referred to as *small* workflow.

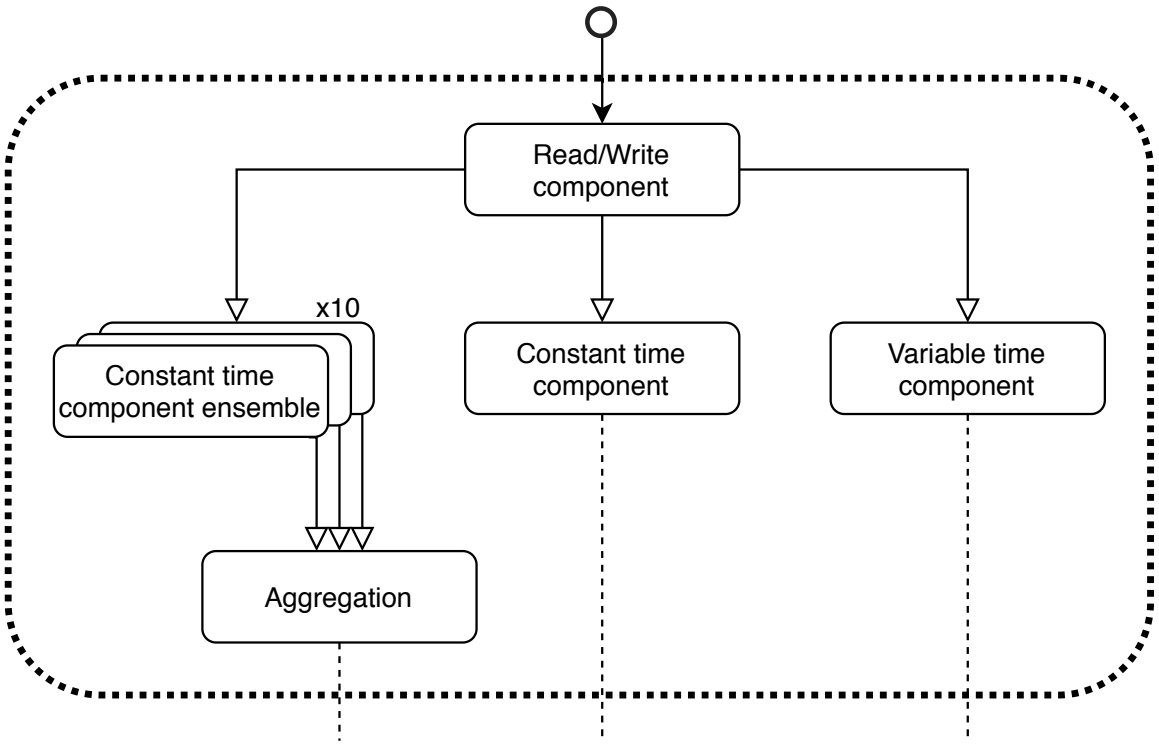

Figure 2.1: All components and their dependencies for the example sandbox workflow deployed here.

2.1.3.1 Read/Write component

First of all, one task is required that both reads and writes data.
The task's code is as simple as the following few lines:

```
1 input_path = sys.argv[1]
2 output_path = sys.argv[2]
3 os.makedirs(os.path.dirname(output_path), exist_ok=True)
4 shutil.copyfile(input_path, output_path)
```

Listing 2.1: read_write.py

After making sure all the directories in the output path exist in line 3, the input data will be copied to the output path, effectively reading in the data and instantly writing it again.

2.1.3.2 Send/Receive component

Secondly, a data broker task is required to simulate sending and receiving data across tasks. This task will be run multiple times in the workflow, concretely each time one

task, that is not a data broker, finishes its execution.
In this implementation of the workflow, the communication uses a *push* concept. A task will push the generated data so any other task can use it, actively offering their data without having any task demanding it, not even knowing whether the data will ever be used by any other task. The opposite would be a *pull* concept where tasks actively demand data instead of only taking what they are getting offered. Data that a task wants to pull can be generated by any task, still preserving the task-independence of asynchronous workflows. Both concepts can be integrated quite well in asynchronous scheduling and both have different advantages. In a perfectly designed scheduler, both concepts can even coexist and be used simultaneously. However, implementing the pull concept does require the scheduler to implement some more advanced features as well, such as runtime estimation, as a task demanding data will have to wait for another task generating this data getting scheduled first, being directly followed by the task that demanded that data in a perfect scenario. That adds a lot of complexity to the scheduler, which does exceed the scope of this work.
As the source code for the data broker task is a lot more complex compared to the other tasks, it will therefore not be examined in detail in this section. Instead, the source code can be found in the appendix in chapter A.

2.1.3.3 Constant time component

Thirdly, a task emulating a component utilizing an exact algorithm will be deployed. Leaving the import statements out, the code for the task looks as follows:

```
1  input_path = sys.argv[1]
2  output_path = sys.argv[2]
3
4  open(input_path, 'rb').close()
5
6  time.sleep(30)
7
8  if comm.Get_rank() == 0:
9      os.makedirs(os.path.dirname(output_path), exist_ok=True)
10     open(output_path, 'wb').close()
```

Listing 2.2: constant_time.py

The task will be run using 12 cores on a single node. However, in this example scenario, there is no actual parallelization, so using more cores would not speed up the process. After reading in the input data, the task will sleep for a constant amount of time, 30 seconds to be exact, before writing some output data which in this case is just an empty file as this data will not be used anymore by any other task.

2.1.3.4 Variable time component

Leaving the import statements out, the code for the task simulating an iterative algorithm looks as follows:

```
1  input_path = sys.argv[1]
2  output_path = sys.argv[2]
3
4  comm = mpi4py.MPI.COMM_WORLD
5
6  time_to_sleep = 0
7  if comm.Get_rank() == 0:
8      open(input_path, 'rb')
9      time_to_sleep = 5*(random.randrange(10)+1)
10
11 time_to_sleep = comm.bcast(time_to_sleep, root=0)
12
13 time.sleep(time_to_sleep)
14
15 comm.Barrier()
16
17 if comm.Get_rank() == 0:
18     os.makedirs(os.path.dirname(output_path), exist_ok=True)
19     open(output_path, 'wb').close()
```

Listing 2.3: variable_time.py

The task will be run using 48 cores on two nodes. Parallelization is realized using the python bindings of MPI, however there is no actual parallelization implemented, therefore increasing the amount of parallel processes does not speed up the execution. After reading in the input data, all processes of the task will sleep for a random amount of time, 5 seconds times a random number between 1 and 10 to be exact, before writing some output data which in this case simply is an empty file again.

2.1.4 Running ensembles

Many ESM workflows do also run so-called ensembles. Multiple instances of a component are run with slightly different parameters forming a component ensemble. The ensemble runs concurrently to the regular version of the component and will only run once.
Ensemble runs are triggered by special events, e.g. some anomaly in the data that might indicate extreme events like a hurricane. Therefore, unlike their regular counterparts, they do not repeat execution after finishing but are run only once.
The sandbox workflow does also simulate such an ensemble run by running the constant time task multiple times, 10 times to be exact. An ensemble run will always

be finished up by the run of an aggregation task using the output of all ensemble members as input.

2.2 Master-worker parallelization

For this work use of the *Hub-and-Spoke* paradigm [9] has been selected which is better known as *master-worker* parallelization pattern on HPC systems. A so-called master has control over a set of workers, which in this work are called tasks.
The master process does not only schedule task execution, but also manages all communication in the workflow. Tasks will run independently from each other and have no information about other tasks that are running. Thus they cannot direclty communicate with other tasks. Communication between tasks will be handled using data. The master is also responsible for data management, which includes tracking available memory space on all nodes of the allocation as well as moving data across nodes.

2.2.1 Structure

There are three main types of processes. First, the master process which will be started as main process scheduling the execution of all tasks. Second, a task process which represents any kind of worker, e.g. an ESM component or a visualization. Lastly, a data brokering process which will be started after the execution for each task in case that there are any other tasks interested in the data generated by that task.
Tasks will be specified using the human-readable data-serialization language *YAML* [16] with a specific structure. These specifications include properties like package requirements, program path, dependencies, etc. A task combines the actual application that will be run, the data input it needs and the output it generates. This allows having data dependencies between different tasks, while a task can depend on an unlimited amount of data generated by other tasks which this application uses as input data.
A data broker is a special kind of task that analyzes generated data after each run of a task and notifies the master if any other tasks are interested in this data using predefined analysis scripts provided in the dependencies section of the task's specification file.
A broker will always be started by the master process itself on the same node that the corresponding task ran on. The result of the analysis which will be run using the aforesaid data will be a boolean value that indicates whether the checked task will be added to the queue using this data as input.
Depending on the data there might be an opportunity to run an ensemble of tasks. If a task should be able to be run as ensemble, a different analysis has to be provided to check if a task should not be executed normally but in an ensemble instead. It is also required to provide a method to prepare data for the different runs of the ensemble.

The master process is also responsible for keeping track of the corresponding nodes a task runs on. Next to the task itself, the master will also store a pointer to the data generated by any process in case it is needed by any task in the future. All data will be managed by the master process and removed on all nodes as soon as it is not needed by other tasks anymore. This does introduce a strong factor influencing the scheduling as each node will have a maximum amount of memory that can be used for data. The closer a node gets to that threshold, the higher the scheduler prioritizes running tasks that will use the data on that node to free memory again.

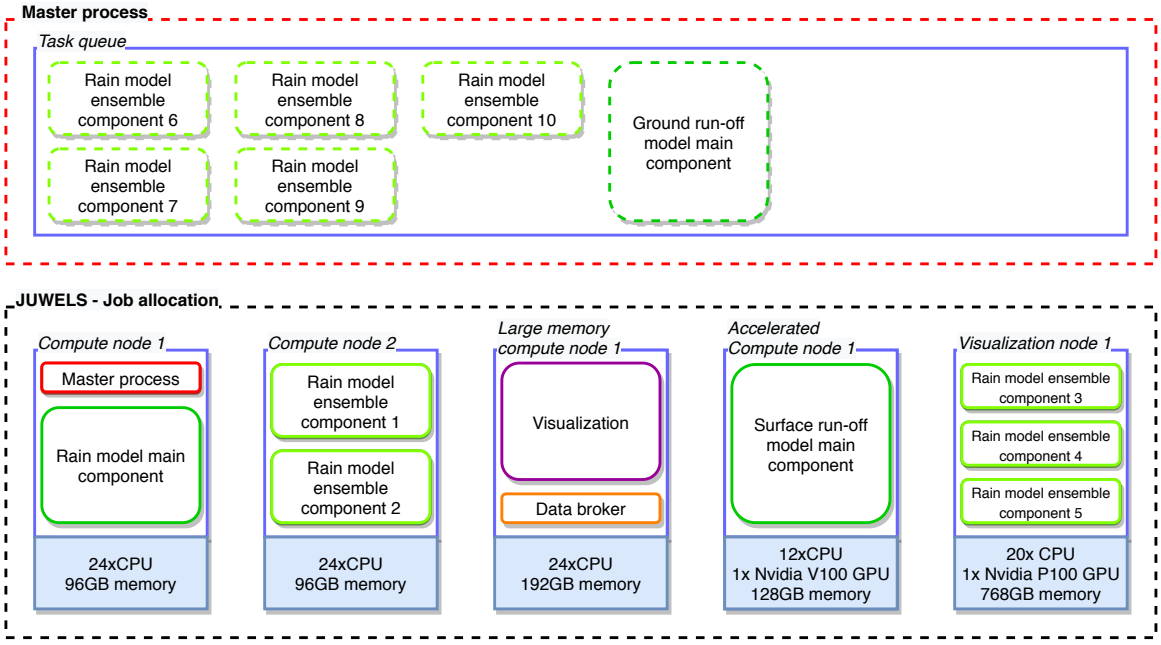

Figure 2.2: A snapshot of some potential ESM tasks distributed on different nodes of a heterogeneous job allocation on JUWELS as well as in the task queue which contains tasks that are currently getting scheduled.

2.2.2 Communication

There is no direct communication between tasks themselves or between a task and the master process. The master will start a task, add the task to the list of currently running tasks, wait until it has finished and potentially generated some data, and afterwards remove the task from the internal list of running tasks. Finally, after the end of the task's lifetime, a data broker will be initialized. During the whole lifetime of the task, there is no communication between it and the master at all, not even initially. Tasks will be initialized using the task specification in form of the provided YAML file for each task and the input data the task is interested in. As soon as the task finished after potentially writing its output data, this data will be used to trigger

the execution of new tasks, altogether requiring no communication.
There still is some communication required though, namely between the master process and data brokers as well as between the master process and the user making use of the *command line interface* (CLI) that has been created within the scope of this work.
As the master process will participate in all possible communication, a straightforward *client-server* approach could be used to handle communication. As current industry standard for intra-datacenter communication, gRPC has been used for all communication purposes.

2.2.2.1 gRPC

The open-source high performance *remote procedure call* (RPC) framework gRPC is a Cloud Native Computing Foundation incubating project developed by Google [17]. It makes use of protocol buffers, a fast binary serialization toolset and language. Remote procedure calls cause execution of subroutines in a different address space, e.g. a different compute node. Many RPC systems use so-called services to define the methods that will be called remotely as well as the data structure used to transmit data to the remote system. In gRPC these services are defined using so-called protobuf files, describing both the service interface and structure of the payload messages using the protocol buffer interface-definition language.
Here gRPC is used to enable communication between the master process and data broker as well as the CLI process. Therefore, two seperate services were defined.
First, a service being responsible for communication from and to data brokers. This service defines two procedures, the first one being called to initialize the data broker, the second one to report the analysis results back from the broker to the master service.
Second, a service sending commands to the master process from the command line. Several procedures were defined in this service, while each of these procedures sends a message with varying payload to the master process and in some cases gets back some information, e.g. the number of tasks currently in the queue.
The communication from a command line interface to the master process enables modification of the ESM during runtime by adding or removing specific components for instance. It could also be used to insert a visualization task to get an overview about the current state of the model.

2.2.3 Data management

The master process manages all data that is being generated by tasks. This includes tracking available memory space on all nodes of the allocation and moving data from filling up nodes to shared storage as well as preloading data on nodes where data will soon be needed by a task.

Each node has a fixed amount of available memory which will be used both to cache data for future tasks as well as for tasks that currently run on that node. A fixed amount of the node's memory will be designated as cache. The threshold is given by a fixed percentage of the total memory of the node and should never be exceeded. In case the memory cache overflows, the master process will have to move some data to a shared storage and load it onto a node again at a later point in time. However, this should be prevented as much as possible. For maximum efficiency the scheduler will attempt to minimize data movement by keeping as few data in the cache as needed by prioritizing tasks that work with data on nodes approaching the threshold so cached data can be removed again.
If data is expected on a node where it is not yet available, the data should be moved or copied to that node before the actual task requiring that data starts running on that node. Preloading data efficiently can become very complex and will therefore just be implemented in a very basic way in the scope of this work, namely simply preloading the data as soon as the task execution starts. For it to work without introducing a lot of idling however, accurate runtime estimation is required. While data is loaded onto a node, another task can run on that node in the meantime. In the best case, both the IO operation and task execution should take a similar amount of time to prevent idling and amassment of data that is not currently used on a node.

2.3 Scheduling

The mathematical problem that has to be solved to perfectly schedule tasks across multiple nodes seems to be a quite common problem at first. However, there are very few exact solutions for this type of problem. As no algorithm exists that can solve this problem in polynomial time, many people use heuristics to get close to an optimal solution. While a near-optimal solution would still be acceptable, there is no existing solution that solves this specific problem. To be more concrete, the problem to solve is called a *multiple variable size knapsack problem with item fragmentation* or as bin packing problems are more common a *bin packing problem with constraints with variable bin size with precedence with fragmentation.* There are lots of solutions for most sub-problems like the knapsack problem with fragmentation, however when adding more constraints, there doesn't seem to be a solution that solves all problems. Therefore, the solution was to either develop a completely new algorithm or to implement a simple scheduling algorithm that either only solves parts of the problem or finds a solution for this problem not necessarily delivering a good solution in all scenarios. For testing purposes a semi-optimal algorithm has been used within the scope of this work.

2.3.1 Formal definition

We are given a list I of m items and a set B of n bins. Each item $i_k \in I$ has a size $s(i_k) \in \mathbb{Z}^+$ and each bin b_l has a capacity $c(b_l) \in \mathbb{Z}^+$. The goal is to maximize the accumulated size of all items packed into bins under the condition that the summed size of items in a bin does not exceed the bins capacity. An item does also have a profit p_k assigned to it which results in another objective, namely to maximize the accumulated profit of all items packed into bins. Items can be fragmented equally dividing its size. Fragmenting a task does however add an overhead, therefore reducing the profit of the task by a constant value r for each fragment.

2.3.2 Sample scheduler

In this scenario nodes are the equivalent of bins and tasks the equivalent of items. Each node has a maximum capacity that is given by the number of cores available on that node. The size of a task is given by the number of MPI-processes that will be used to run the task multiplied by the number of cores used by each process. The maximum number of fragments that can be created of the task is therefore given by the number of MPI-processes for the task. Tasks have a priority value that corresponds to the profit of items in the formal definition of the problem.
As scheduling heuristic, a standard next-fit algorithm [18] has been implemented for this sample scenario and was further improved to support multiple nodes (bins) as well as precedence. The algorithm does not necessarily find the best possible solution and other heuristics [**heuristics**] might be utilized to improve scheduling performance. However, the implemented heuristic still schedules tasks successfully according to their priority while trying to use as many resources as possible, therefore achieving sufficient node utilization in this sandbox scenario.

2.4 Benchmarks

Several benchmarks will be used to evaluate the different aspects of this asynchronous scheduler for ESM workflows such as performance, memory usage and flexibility.

2.4.1 Performance

To evaluate the performance aspect of the scheduler, statistics functionality has been implemented, gathering information about the individual tasks. Concretely, for each task it saves the time it entered the scheduling queue and measures the time it waited in the queue as well as the time its execution took. Additionally, the total lifetime and the node each task ran on will be output.

In order to generate statistics for evaluating the sandbox scenario's capabilities of appliance in a real-world scenario with actual ESM components, the workflow described above in section 2.1.3 was run utilizing varying numbers of nodes, one to eight to be exact, in the Slurm job that was started for each run of the workflow.

When it comes to comparing the performance of the asynchronous scheduler to a fictional synchronous version, the best case execution times were calculated for the synchronous scenario of the workflow to compare these times to the execution times of the asynchronous scenario which was measured using the Unix built-in `time` command.

2.4.2 Memory and IO efficiency

As described in sections 1.3.3.2 and 2.2.3, generated data will be stored in on-node memory to reduce the time another task takes to load this data. If the task requiring the data runs on the same node, the IO time implied by loading the data can be reduced to the bare minimum.
To get an estimation for the total IO time of the workflow, the times data is moved across nodes will be counted for workflow runs using one to eight nodes.

2.4.3 Scalability

There are two types of scalability that are used to characterize the efficiency gain of a parallel workflow.
First, there is weak scalability which is the result of increasing the amount of available resources without altering the problem size.
Second, strong scalability is defined by the change of execution times when increasing both available resources and the problem size.

2.4.3.1 Weak scalability

The weak scalability of the sandbox workflow is determined by scaling up the number of available nodes from one to eight nodes without altering the workflow described in section 2.1.3.
Multiple time statistics will be gathered for each individual task to see how exactly an increased number of nodes affects the execution times of individual tasks and the overall performance of the scheduler.
For each task, the time a task entered the queue relative to the start of the workflow, the queue- and runtime as well as its overall lifetime will be measured.
Afterwards, the number of tasks that run concurrently in a specific time interval are

counted and compared when running with a single and eight nodes.

2.4.3.2 Strong scalability

For the strong scalability benchmark, two more workflows will be tested, both with a larger size than the workflow presented above. For this benchmark, these two workflows as well as the existing one presented above will be run using one to eight nodes. Both workflows use the same components as the smaller workflow, but use each component twice instead.

The first scaled workflow as shown in figure 2.3 does not increase its complexity, which means the amount of dependencies between tasks stays the same as for the smaller workflow. In the following chapters, this workflow will be referred to as *large* workflow.

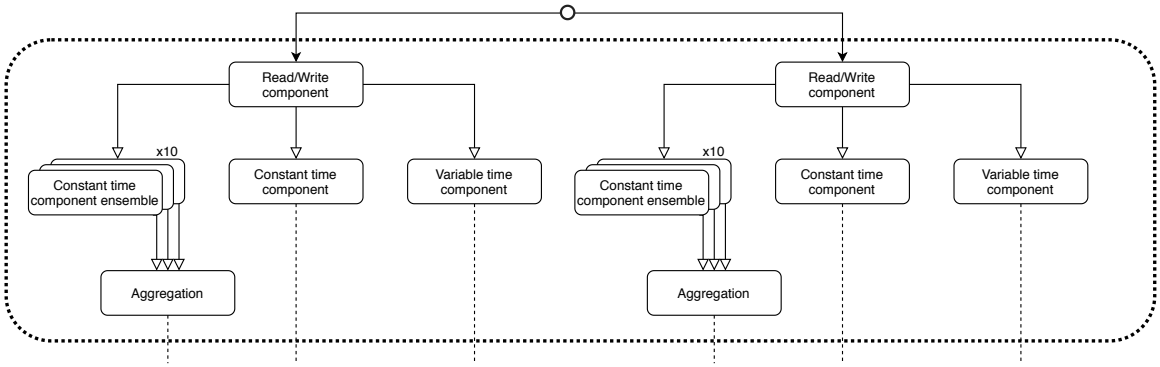

Figure 2.3: A scaled version of the sandbox workflow without additional dependencies between tasks.

The second scaled workflow as shown in figure 2.4 does also increase its complexity by adding more dependencies between tasks. In the following chapters, this workflow will be referred to as *large (complex)* workflow.

2.4.4 Flexibility

As an indicator for flexibility of the workflow, the node-sharing capabilities will be evaluated. These can be evaluated in the same way as for the weak scalability benchmark described in section 2.4.3.1 above. The times each task started running as well as the runtime will be aggregated to check if or if applicable how many tasks are running on a node simultaneously.

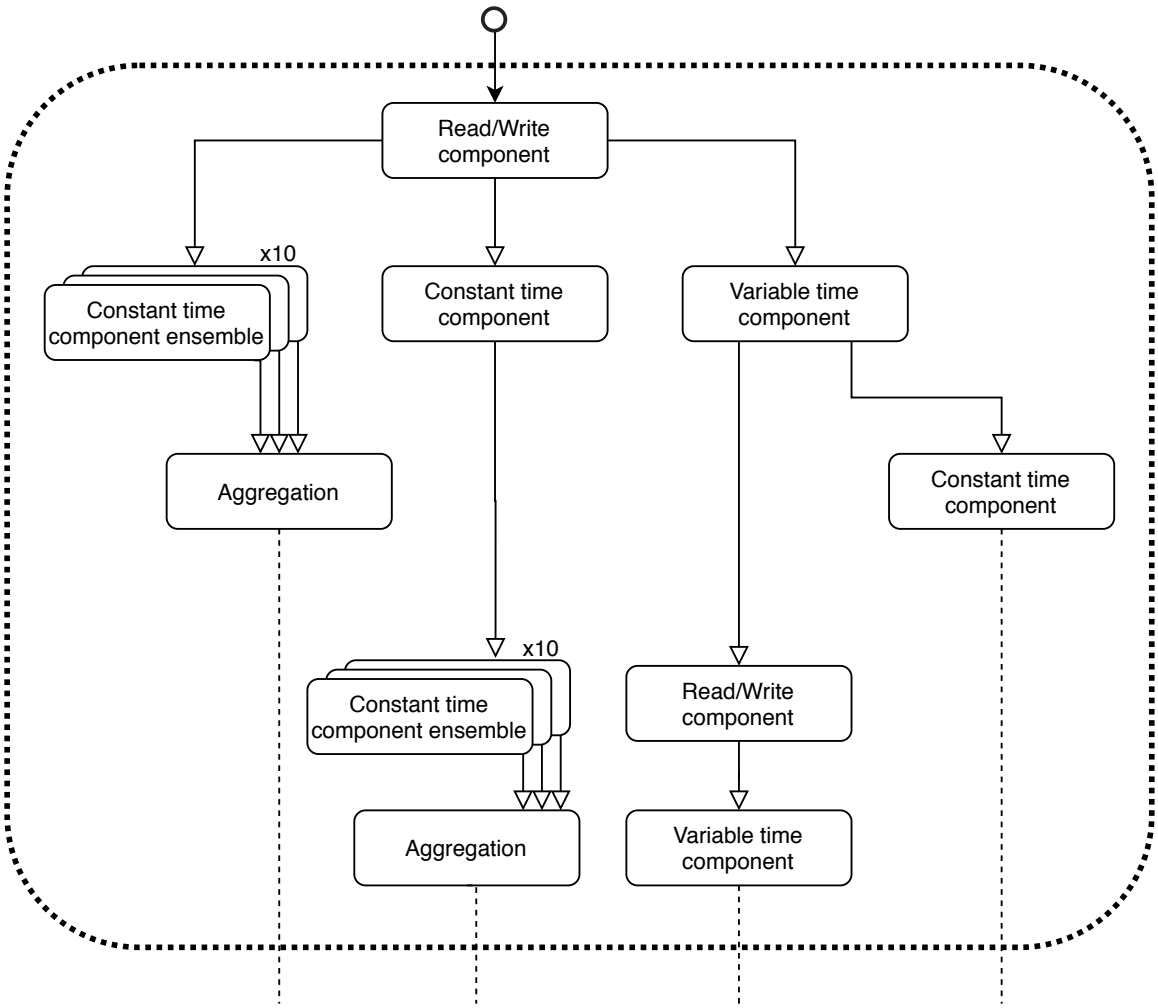

Figure 2.4: A scaled version of the sandbox workflow with additional dependencies between tasks.

2.4.5 List of benchmarks

Benchmark	Workflows used	Description
Performance	Small [1-8 nodes]	Measuring execution times of the asynchronous version and comparing these times to theoretical values for a synchronous version.
Memory and IO efficiency	Small [1-8 nodes]	Counting the number of times data is moved across nodes to evaluate the IO time reduction implied by the optimizations that were made.
Weak scalability	Small [1+8 nodes]	Measuring execution and queue times of individual tasks and comparing them when running with one and eight nodes respectively.
Strong scalability	Small, Large, Large (complex) [1-8 nodes]	Running three different workflows and comparing their execution times.
Flexibility	Small [3 nodes]	Measuring execution and queue times of individual tasks using three nodes and evaluating concurrency of tasks.

Table 2.1: List of all benchmarks with the respective workflows used by each benchmark.

3 Results

3.1 Synchronous vs. asynchronous performance

The asynchronous version of the workflow has been run on the JUWELS supercomputer using one to eight compute nodes.
To compare the performance of the asynchronous version, runtime values for a hypothetical synchronous version were calculated by adding the individual runtimes of each component in the workflow as follows:
For the runtime of the *read/write* and the *aggregation* components, the average value of the asynchronous runs were used. Both the *constant* and *variable time* component runtimes were estimated by using the maximum time they could take which is 30 seconds for the constant time component and 50 seconds for the variable time component. This estimation is based on the fact, that the synchronous scheduler does schedule all tasks before execution which does not allow tasks that depend on these tasks to be scheduled before their execution ended. For both algorithmic components (variable and constant time), a 0.1 seconds extra time for IO operations has been added as this turned out to be the average IO time for all operations performed in this example scenario.
The ensemble which was run as part of the workflow consisted of 10 ensemble members and the aggregation task. The ensemble members have been represented by the constant time component, therefore running the constant time component ten times in terms of the ensemble and once regularly as part of the workflow.

Component	Runtime	MPI processes	CPU threads
read/write	4.0	1	12
Variable time	50.1	2	40
Constant time	30.1	1	20
Aggregation	1.5	1	20

Table 3.1: Runtime in seconds, number of spawned MPI processes and number of CPU threads per process for all components.

To estimate the total time the synchronous version of the workflow will take, we will assume the architecture, or more concretely the available nodes, will be used as efficiently as possible.
When there is only a single node available, the order in which the components run

is irrelevant. First, the read/write component will run for 4 seconds. Afterwards the iterative process will be started, using 80 of the available 96 physical CPU threads on the node running for 50+0.1 seconds. This component will be followed by the ensemble, running two full batches of 4 tasks each, in total utilizing 80 threads twice, while each batch takes 30+0.1 seconds until finishing execution. The third batch will only run two ensemble members which leaves enough resources to also run the regular constant time component, also running for 30.1 seconds. Finally, when all ensemble members finished their execution, the aggregation component is run, taking 1.5 seconds.
In total, the runtime adds up to $4 + (50 + 0.1) + 3 * (30 + 0.1) + 1.5 = 145.9$ seconds. When instead of a single node there are two nodes available, the ensemble can start running on the second node while the iterative component still runs on the first node. The third batch of the ensemble run and the regular constant time component will then start on the first node after the variable time component. This allows the aggregation component to start running after $4 + 50.1 + 30.1 = 84.2$ seconds. In total, in this scenario the runtime adds up to $84.2 + 1.5 = 85.7$ seconds.
The runtimes for other numbers of nodes can be calculated the same way and are visualized in figure 3.1.

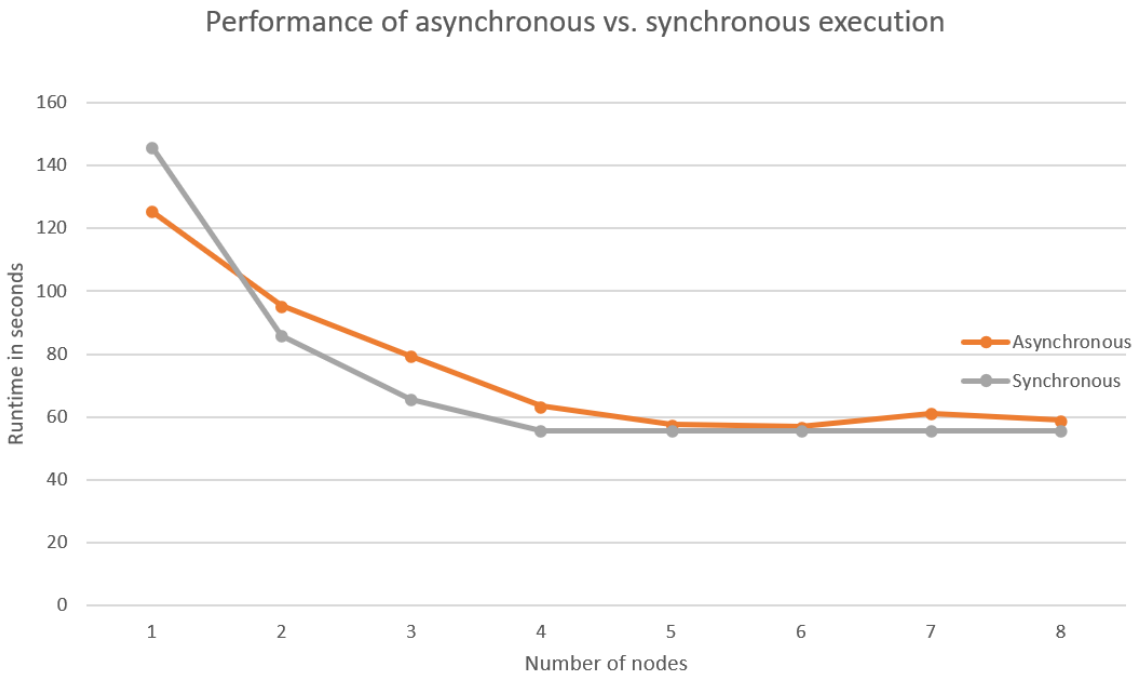

Figure 3.1: Performance of the asynchronous vs. synchronous version of the workflow.

Figure 3.1 shows the measured performance of the asynchronous version of the workflow compared to the hypothetical optimal synchronous performance. While the

ideal synchronous performance is better for more than one node, the single-node performance of the asynchronous version is remarkably better. When using many nodes, the performance of both versions seem to almost converge.

3.2 Memory and IO efficiency

Strong emphasis has been laid on reduction of IO time by managing data as efficiently as possible. To reduce data movement, generated data is temporarily stored in the memory of the node it was produced on, making it available for future tasks running on that node without any additional IO time in between. To quantify the actual advantage implied by on-node data storage, the times data was moved across nodes has been measured.

Number of nodes	Data shared across nodes
1	1
2	8
3	12
4	13
5	13
6	14
7	13
8	16

Table 3.2: The number of times data has been copied from one node to another relative to the number of nodes.

Table 3.2 shows the times data was shared across nodes which is proportional to the IO time of the workflow. Therefore, the fewer times data has been shared, the lower the IO time of the workflow has been.
The minimum IO time was therefore reached for a single node, where data has only been loaded onto a node once, when initially loading data to the very first component, namely the read/write component.

To evaluate the effective improvement of on-node data caching, the measured values have to be compared to the default case without this optimization. In the default case, data will always be stored in a centralized storage and will be copied on a node each time a task needs the data. The number of times data has to be loaded into memory is equivalent to the number of tasks using any data as input, which in most cases will be the total number of tasks. In this sandbox scenario, there are 28 tasks loading data.

When now comparing this value to the results shown in table 3.2, the effective improvement can be calculated for each number of nodes utilized. For two nodes, there would be 3.5 times more data loads required if no optimization would have been used and for 6 nodes there still was an improvement of about 50%.

3.3 Scalability

3.3.1 Weak scalability

As shown in figure 3.2, when running the small workflow using 8 nodes, no task is run in the first 10 seconds while two tasks were run already when only using a single node. In the following time intervals however, many more tasks, up to 11 more to be concrete, were run when using 8 nodes compared to the execution using a single node.
On average, about 10 tasks were run concurrently when using eight nodes while only 5 were run for the single-node configuration.

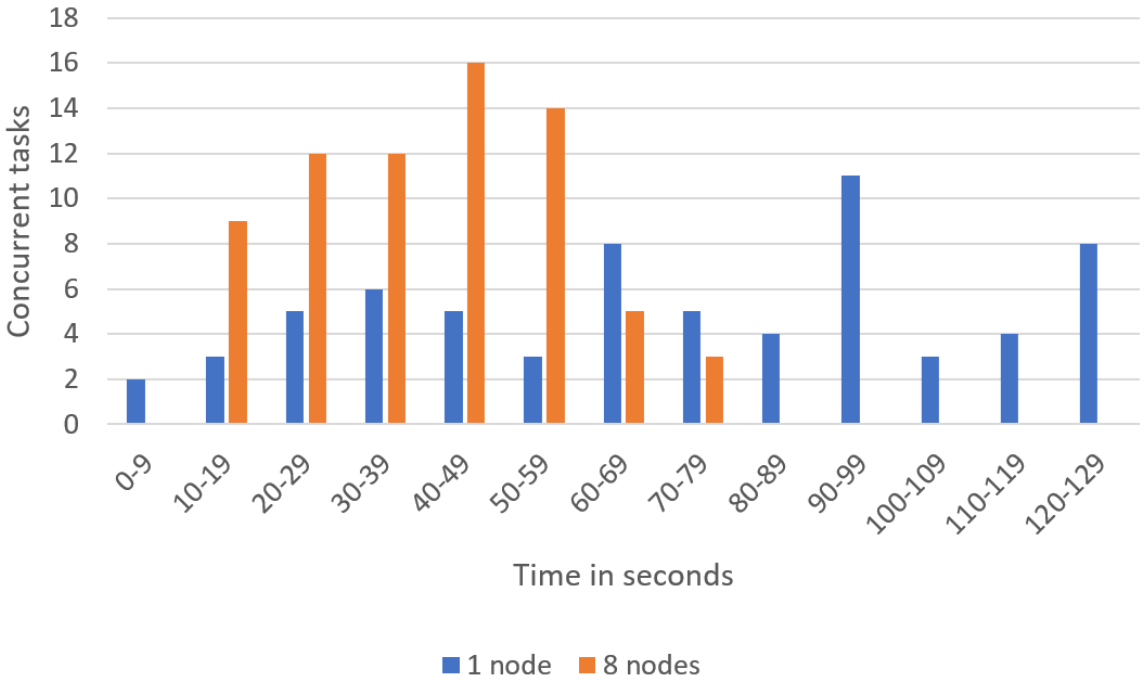

Figure 3.2: Tasks running concurrently on 1 and 8 nodes.

Multiple tables listing detailed information about each task can be found in chapter 5 in the appendix.
The starttime column of these tables presented there shows the point in time the task

entered the queue relative to the start of the workflow. Adding up the values in the queue- and runtime column for each task gives the value of the total lifetime column. The last column tells about the node a respective task ran on.
When comparing the values for one and eight nodes in table 5.1 and 5.3 the most significant difference can be found in the queuetime column. The time some components spent in the queue was much longer than the time they spent running when using a single node, while for the eight-node configuration almost all tasks had either an insignificant queue time or one much lower than their runtime. The ensemble aggregation presents the only task that had a fairly long queue time for both configurations.

3.3.2 Strong scalability

Remarkable differences in runtimes were measured for the large and large (complex) workflows as shown in figure 3.3. When increasing the complexity of the large workflow by adding more dependencies, the workflow took about 2.5-4.5 times longer to finish execution.
However, for an increasing number of nodes, the more complex variant of the scaled workflow achieved the highest speedup with a value of 5.4 for 8 nodes. The less complex large workflow did only achieve a speedup of 3.9 which, however, still was about twice as much as the speedup of the small workflow with a value of 2.1.
In contrast to the small version and the complex variant of the large workflow, the large workflow almost approached the ideal curve when using two or three nodes.

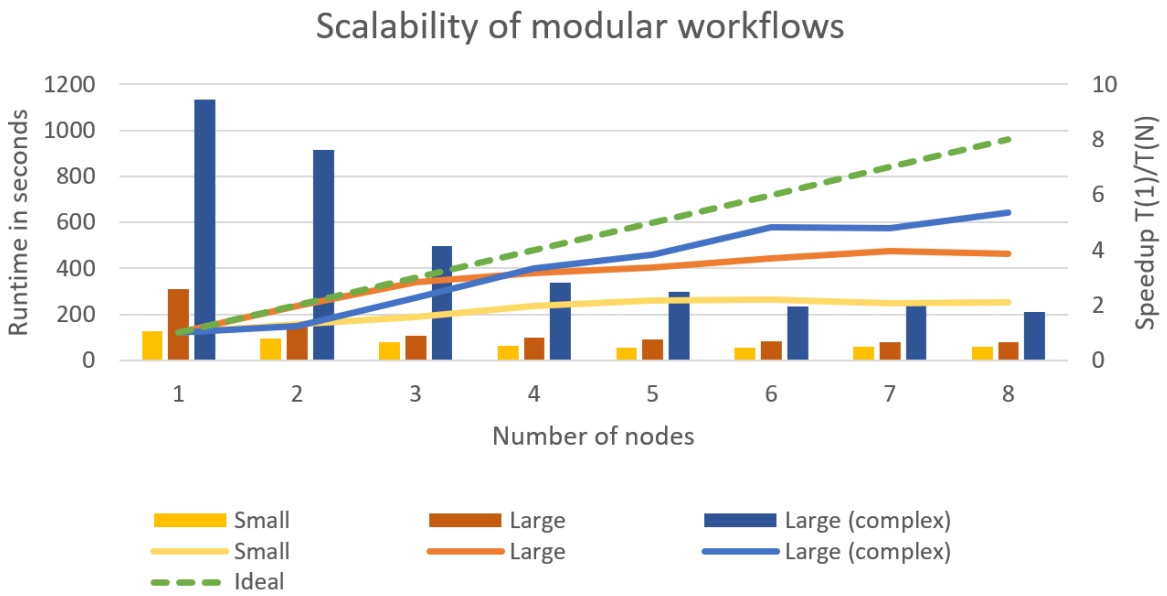

Figure 3.3: Bars showing runtime in seconds and lines representing speedup of three workflows using a varying number of nodes.

3.4 Flexibility

Figure 3.4 shows multiple tasks running concurrently when utilizing three nodes. In the time interval ranging from about 10 to 40 seconds the most tasks were run simultaneously with times where 9 tasks are running at the same time. The variable time algorithm task was split up and therefore ran on 2 different nodes.

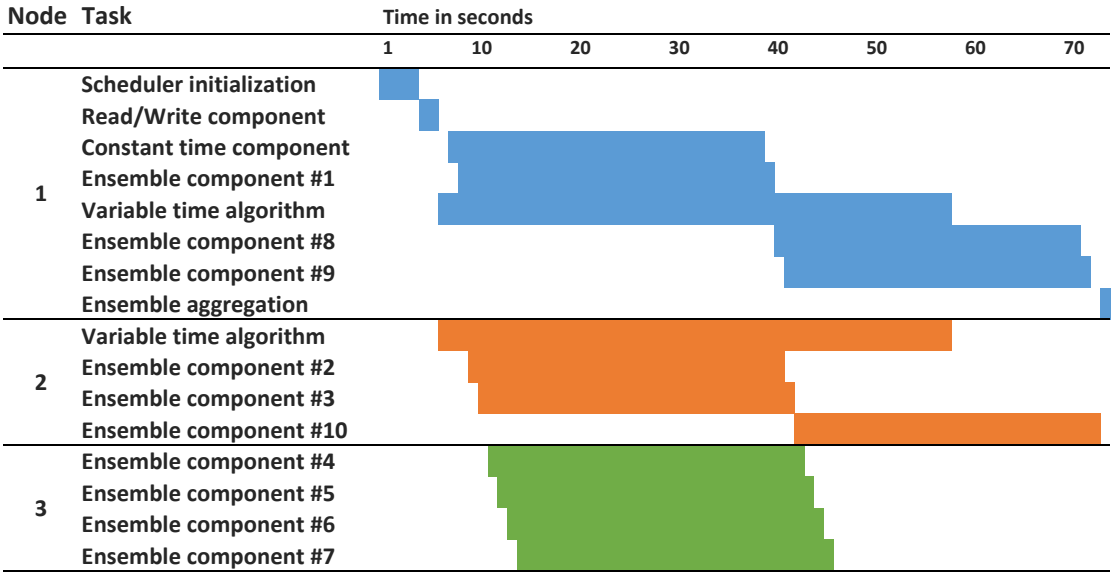

Figure 3.4: Schedule of the workflow using three nodes each represented by a different color.

Detailed time information for each task are listed in table 5.2 in the appendix.

4 Discussion

4.1 Overhead of asynchronous scheduling

Compared to the synchronous scheduling approach the asynchronous one shows a slightly worse performance of the asynchronous one for most node configurations as shown in figure 3.1. While the asynchronous scheduling will most likely be faster in more complex scenarios where it is difficult to synchronously schedule efficiently, here, in the simple example sandbox scenario, the additional overhead makes a noticeable difference in performance compared to the synchronous one that ideally has zero overhead at runtime.
The overhead mainly comes from initialization by analyzing the architecture and setting up the workflow and therefore increases when using more nodes. This can be seen when comparing the start times of the very first task of the run with one node (table 5.1) and the run with eight nodes (table 5.3) which shows a 3.5 seconds later start time for the first task when using eight nodes. Although there undeniably will be increased overhead when scaling up the allocation size, the overhead will be decreased in relation to the total runtime when also scaling up the size of the workflow.

4.2 Scalability

Section 3.3.2 shows two possible ways to scale up the workflow. It is possible to either just add additional components to the workflow without adding extra dependencies which increases the size but not the complexity of the workflow or to scale up both which adds a lot of complexity for each additional dependency between the components.
In reality, each component will also add additional dependencies as in an ESM all components typically are tightly coupled. Using an asynchronous scheduling approach does not only enable unlimited scalability when it comes to the size of the ESM, but even performs really well for large and especially complex workflows as seen in figure 3.3, where the complex version of the large workflow had a higher speedup value than the less complex large one. For a synchronous version one would expect the opposite, the more complex the problem the more difficult it will be to efficiently schedule all components, especially with increasing node allocation size and when using a heterogeneous node allocation.
While scalability is theoretically unlimited when using asynchronous scheduling, it is effectively limited by the scalability of the workflow itself, or more precisely by the

scalability of all components that can run concurrently. If at one point in time only few tasks can be run simultaneously due to the chosen dependencies, the scheduler might be able to run all tasks using the maximum amount of resources possible for these tasks without fully utilizing the upscaled node allocation.
The scalability of an asynchronous scheduler is, of course, also limited by the efficiency of the scheduling algorithm itself. As already stated, an increasing problem size also increases the overhead of the scheduler. Solutions to this challenge will be discussed in section 4.4.

4.3 IO time optimization

The IO times could already be reduced by over 50% by reusing data stored on on-node memory instead of storing all data on centralized storage like the scratch space on Juwels ($SCRATCH). This could even be further optimized by completely relying on on-node memory altogether. To exchange data across nodes, the current implementation uses the scratch space as intermediate stop before copying data on another node because *remote direct memory access* (RDMA) is not available on the Juwels supercomputer. An alternative to using RDMA would be using MPI to share data between two tasks. Concretely, the MPI buffer on the receiving side needs to be set to a memory address pointing to the space that is mounted as a temporary filesystem on `/dev/shm`. That way, data will be copied directly from the memory on one node to the memory of another node without having to explicitly move any data. MPI has been optimized for most modern architectures, making up for maximum efficiency. Such optimizations are however beyond the scope of this initial study.
When completely relying on on-node memory the total memory capacity might not be sufficient for all the data that will be produced throughout the run of the workflow. When temporarily storing data on a node it has to be guaranteed that tasks that currently run on that node will have sufficient memory capacity available for the data they are producing. If the data size on a node comes close to the maximum capacity it will be required to move some data on a centralized storage with a higher capacity such as the scratch space on Juwels. While the HPST storage system on Juwels functions as a cache-like storage that now allows directly writing to or reading from it, in theory low-latency storages with moderate capacity would be the best choice to temporarily store data that had to be moved from on-node memory due to overflowing memory. Many supercomputer architectures offer such a storage, in the end its a choice of design whether such low-latency storages are used as an option to allow developers to optimize their applications or as a form of cache that automatically optimizes loading times in many scenarios.
Another optimization could be to preload data on nodes where it will be needed by future tasks running on that node. This would effectively reduce IO times to zero, as there would be no time anymore where tasks have to wait for data movement. To preload data efficiently it is required to accurately estimate the runtime of all tasks to

see where data will be needed exactly at which times in the future. As the resources could theoretically be assigned dynamically at runtime as well depending on the resources currently available, this has to be considered as well when estimating runtime. A possible way of solving this complex problem would be to use a machine-learning based scheduler which will be discussed later in section 4.4.2.

4.4 Scheduling aspects

4.4.1 Exact solution vs. heuristics

Using an exact solver as scheduling algorithm can be problematic in large scenarios when there are many tasks in the queue at some times. As no exact solver exists that only scales with $O(m^p)$, overhead implied by scheduling will easily become unacceptably high.
Therefore, using heuristics instead of exact solvers offers a reasonable solution in most scenarios. A well-designed heuristic will be able to come very close to an optimal solution and will even work extremely well although not finding the best possible solution. While this ultimately depends on the actual problem, in many cases specific group of tasks can be scheduled interchangeably. If an ESM consists of a single *directed acyclic graph* (DAG), tasks have to be scheduled in a predetermined order, however, if it consists of multiple independent DAGs the scheduler can decide which tasks of these graphs to run first. Eventually, all DAGs have to be run entirely.
Apart from the scheduling algorithm itself, the calculated priority of a task determines which tasks will be run first. The priority of a task takes many different factors into account, such as the task's size and its time in the queue and can therefore be used to add additional rules, e.g. large tasks getting prioritized over small tasks as small tasks can usually be scheduled more easily.
In contrast to typical asynchronous applications which mostly have no state, in ESMs state is especially important, as each component alters the current state with its computation and generated data. The timestep data belongs to must play a key role in scheduling as components should be prioritized that calculate data for a time period close to the current state. In asynchronous scheduling scenarios the calculation does not necessarily have to be strictly linear in time. The scheduler might decide to first run components that calculate data further in time, before eventually coming back to tasks calculating present time steps as this scheduling behavior might increase efficiency in some scenarios. Moreover, it would even be imaginable to calculate some predictive data for some component that allows running subsequent components depending on this data. In the meantime, this data would be reassured and, if necessary, updated which would result in a cascade-like behavior, where all dependent tasks would be re-run using the updated data. In many cases the predicted, lower accuracy data will be sufficient for all dependent calculations which in the end saves valuable computation time. The flexibility of asynchronous scheduling

allows making such dynamic decisions at runtime without any drawbacks, while for synchronous scheduling such a scenario would practically be impossible to realize as the number of performed calculations has to be strictly known beforehand.
There are different solutions to enforce a different scheduling depending on the priority. One example would be a high-priority queue with all tasks with a priority above a certain threshold. This high-priority queue would have to be completely scheduled before the regular queue would be included into the scheduling again. This threshold value could even be determined dynamically, for example at a priority value where there are only a few tasks with much higher priority than all other tasks.
Another solution would be to split the allocation into two parts, one for large tasks only and one for all small tasks. However, this might result in an increased IO time again as data locality could sometimes not be used for optimization anymore.

4.4.2 Machine-learning-based scheduler

As already mentioned in the previous chapters, finding an optimal scheduling solution is very difficult as the problem to solve is that complex that no optimal solutions exists yet. Therefore, a machine-learning approach might be used to replace explicit scheduling.
Two possible machine-learning scenarios come into consideration, supervised and reinforcement learning.
In the supervised scenario, data would be generated by existing schedulers that would then be evaluated to produce labeled data which will subsequently be used to train the machine-learning model.
The reinforcement learning scenario would work similarly, but would not need any existing schedulers to work. This would however result in an increased training time. A cost-function would be defined to evaluate the scheduling efficiency which would then be used to run the scheduling model many times, optimizing its efficiency with each generation. For this to work, the scheduler would need some concrete statistics for each task, such as the estimated runtime, which would then be used as input data. This is required to train the model on arbitrary tasks instead of letting the scheduler learn the runtimes of the concrete tasks used for the training. To get a fast and reliable runtime estimation for tasks, another machine-learning model could be trained for example.

4.5 Flexibility

The biggest advantage of the asynchronous scheduling comes from its flexibility which is based on the fact that scheduling is performed completely at runtime. Therefore, it is possible to make changes to the component configuration and instantly test these changes. Additionally, it enables adding or replacing arbitrary components at runtime, thereby allowing for novel ESM applications, which would be difficult to realize

in the traditional ESM set-up.
Next to workflow design, the resulting architecture flexibility of asynchronous scheduling makes earth system modeling much more comfortable as not only the portability of the workflow is assured, but now heterogeneous architectures can be used to their full extent to make use of modern supercomputer development which introduces many ways to accelerate computation such as additional GPUs on nodes while usually, adaptation of model codes to new architectures tends to be a quite demanding development [19]. When making use of synchronous workflows, the modeler himself has to take care of resource assignment when he wants to make use of these accelerators. As for complex workflows there is no way to accurately estimate at which times resources will be available, these accelerators could not be used efficiently. When using asynchronous scheduling instead, this will be handled automatically by the scheduler with much higher efficiency. Even adding support for these accelerators to existing ESM components is simplified using the asynchronous approach as there is no gigantic, monolithic codebase as it would be the case when using the traditional, synchronous approach. Components could simply be substituted by the version with accelerator support after integrating it into its code without having to rebuild the whole model. When trying to modify an existing model it might not even be possible to easily extract components again in the synchronous case.

5 Conclusion and Outlook

The use of asynchronous scheduling enables more flexibility when building and deploying ESM workflows compared to legacy ESM workflows that usually run synchronously. The increased modularity allows running the workflow on heterogeneous architectures that supercomputers tend to adapt lately. This comes at the cost of some performance reduction, which, however, will not be prohibitive under many circumstances as shown by the results from this thesis. Fully optimized synchronous workflows will, of course, run faster on the specific architecture they were built for, due to no additional overhead at runtime. The asynchronous version on the other hand will optimize itself at runtime and therefore runs as efficiently as possible on any architecture without having to optimize the workflow manually. Increased scheduling flexibility furthermore allowed optimizing IO times by reusing data stored on on-node memory, reducing IO times by more than 50%.
Altogether, the improvements made up for high scalability, showing very high efficiency gain for complex workflows when increasing the amount of available resources. This proof of concept therefore showed, that asynchronous scheduling may be able to solve the challenges introduced by the upcoming exascale ESM development.

In the future, it will be interesting to create an actual ESM workflow using the asynchronous scheduler which would really test the flexibility, usability, and performance of this novel approach to earth system modeling.
After creating such a workflow, the performance and usability should be compared to other modern ESM workflows like CESM2 to see if this approach is an actual improvement over other development approaches.
Apart from a function-complete CLI, an API to the scheduler might be provided to get information such as advanced statistics, the current state of the queue, or special events that occurred that all might lead to workflow alterations or even optimization at runtime.
Currently, only slurm-based systems are supported, for full flexibility however, the workflow should be able to run on any architecture and system.
Testing configurations with both GPUs and CPUs was beyond the scope of this thesis. However, it is clear that such configurations can be easily realized using an asynchronous scheduling approach, while they are much more difficult to implement in a synchronous set-up. In an asynchronous scenario, tasks might even give the scheduler the option to use CPUs and GPUs interchangeably, allowing the scheduler to decide at runtime which resources to assign to which task based on what resources are currently available.

Data broker task source code

```
1  # IMPORTS - Start
2  import subprocess
3  import sys
4  import os
5  from shutil import copyfile
6
7  from messaging.grpc_client import BrokerGRPCClient
8  from utils.data import PathManager, DataPointer
9  # IMPORTS - End
10
11
12 def main():
13     if len(sys.argv) < 5:
14         exit(1)
15
16     tasks_path = str(sys.argv[1]) # Path to the directory containing all tasks
17     task_id = str(sys.argv[2]) # Shortid of the task
18     task_node = str(sys.argv[3]) # Node the task generated data on
19     data_path = str(sys.argv[4]) # Path to the data file
20     master_node = str(sys.argv[5]) # Ip address of the master node for gRPC connection
21
22     # Creating a DataPointer object using the command line argument information
23     dp = DataPointer(task_node, data_path)
24
25     grpc_client = BrokerGRPCClient(master_node)
26
27     # Getting all tasks that are interested in this data
28     simple_deps, ensemble_deps = grpc_client.request_init(task_id)
29
30     def eval_dep(dep, requirements):
31         """
32         Run the evaluation for a given dependency
33         """
34         exec_strings = list()
35         for requirement in requirements:
36             exec_strings.append("module load " + requirement + " &&")
```

```
37                exec_strings.append(dep.analysis.replace("INPUT_PATH", data_path))
38            return_code = subprocess.call(" ".join(exec_strings), shell=True, stdout=subprocess.PI
39            return return_code
40
41        interested_tasks = list()
42        ensembles = list()
43
44        # Evaluate all dependencies that might trigger execution of regular (simple) tasks
45        for simple_dep in simple_deps:
46            if eval_dep(simple_dep, simple_dep.requirements) == 0:
47                interested_tasks.append(simple_dep.name)
48                ensembles.append(False)
49
50        # Evaluate all dependencies that might trigger execution of an ensemble
51        for ensemble_dep in ensemble_deps:
52            if eval_dep(ensemble_dep, ensemble_dep.analysis_requirements) == 0:
53                interested_tasks.append(ensemble_dep.name)
54                ensembles.append(True)
55
56        # Send all tasks that are interested (positive evaluations) to the master process
57        grpc_client.send_tasks_interested(interested_tasks, ensembles, task_id, task_node, data_p
58
59
60    if __name__ == '__main__':
61        main()
```

Listing 5.1: read_write.py

Detailed list of task execution times

One node

Taskname	Starttime	Total lifetime	Queuetime	Runtime
Read write	6.467	3.205	2.275	0.929
Broker	9.672	1.487	0.001	1.487
Iterative algorithm	11.112	16.867	0.001	16.866
Ensemble #1	11.112	48.692	16.874	31.818
Ensemble #2	11.112	48.698	16.877	31.820
Ensemble #3	11.112	48.767	16.881	31.896
Ensemble #4	11.112	1:20.572	48.697	31.874
Ensemble #5	11.112	1:20.591	48.708	31.883
Ensemble #6	11.112	1:20.621	48.784	31.837
Ensemble #7	11.112	1:20.927	48.769	31.158
Ensemble #8	11.112	1:52.461	1:20.595	31.866
Ensemble #9	11.112	1:52.467	1:20.623	31.843
Ensemble #10	11.112	1:52.483	1:20.574	31.909
Exact algorithm	13.342	32.739	1.035	31.704
Broker	27.980	0.518	0.003	0.515
Broker	59.811	1.024	0.007	1.016
Broker	59.805	1.038	0.001	1.037
Broker	59.890	1.008	0.001	1.007
Broker	59.880	1.059	0.001	1.058
Broker	1:31.685	0.932	0.005	0.931
Broker	1:31.703	0.917	0.003	0.916
Broker	1:31.734	0.934	0.005	0.933
Broker	1:32.039	0.733	0.001	0.733
Broker	2:03.574	0.844	0.003	0.843
Broker	2:03.579	0.838	0.001	0.835
Broker	2:03.596	0.822	0.001	0.821
Ensemble aggregation	1:46.943	20.165	17.651	2.514
Broker	2:04.841	0.498	0.002	0.495

Table 5.1: Detailed comparison of execution times of all tasks run in the example workflow using one node as well as the node the task ran on.

Three nodes

Taskname	Starttime	Total lifetime	Queuetime	Runtime	Node/-s
Read write	4.443	1.858	0.988	0.869	1
Broker	6.301	9.448	0.003	9.445	1
Exact algorithm	7.660	32.729	1.024	31.705	1
Ensemble #1	7.660	33.714	2.025	31.689	1
Iterative algorithm	7.660	51.713	0.001	51.712	1,2
Ensemble #2	7.660	34.821	3.032	31.789	2
Ensemble #3	7.660	35.796	4.035	31.760	2
Ensemble #4	7.660	36.821	5.041	31.779	3
Ensemble #5	7.660	37.800	6.031	31.769	3
Ensemble #6	7.660	38.782	7.030	31.752	3
Ensemble #7	7.660	39.808	8.043	31.765	3
Ensemble #8	7.660	1:05.495	33.728	31.767	1
Ensemble #9	7.660	1:06.508	34.737	31.770	1
Ensemble #10	7.660	1:07.584	35.830	31.753	2
Broker	40.390	2.067	0.002	2.064	1
Broker	41.375	1.082	0.015	1.067	1
Broker	42.482	1.051	0.000	1.050	1
Broker	43.456	0.536	0.035	0.501	1
Broker	44.482	0.498	0.000	0.497	1
Broker	45.461	0.495	0.004	0.490	1
Broker	46.443	0.509	0.004	0.504	1
Broker	47.469	0.493	0.003	0.490	1
Broker	59.374	0.500	0.000	0.499	1
Broker	1:13.156	0.494	0.003	0.491	1
Broker	1:14.169	0.495	0.000	0.495	1
Broker	1:15.244	0.489	0.005	0.484	1
Ensemble aggregation	1:15.667	0.500	0.001	0.499	1
Broker	1:16.168	0.496	0.004	0.491	1

Table 5.2: Detailed comparison of execution times of all tasks run in the example workflow using three nodes as well as the node the task ran on.

Eight nodes

Taskname	Starttime	Total lifetime	Queuetime	Runtime	Node/-s
Read write	10.030	1.904	1.005	0.899	1
Broker	11.935	12.740	0.001	12.739	1
Exact algorithm	13.342	32.739	1.035	31.704	1
Iterative algorithm	13.342	51.707	0.003	51.704	2
Ensemble #1	13.342	33.750	2.044	31.706	1
Ensemble #2	13.342	34.842	3.076	31.765	1
Ensemble #3	13.342	35.823	4.091	31.732	1
Ensemble #4	13.342	36.916	5.135	31.781	3
Ensemble #5	13.342	37.957	6.169	31.787	3
Ensemble #6	13.342	38.991	7.200	31.790	3
Ensemble #7	13.342	40.045	8.235	31.810	3
Ensemble #8	13.342	41.034	9.254	31.779	4
Ensemble #9	13.342	42.050	10.263	31.787	4
Ensemble #10	13.342	43.016	11.284	31.731	4
Broker	46.082	0.509	0.001	0.508	1
Broker	47.092	0.504	0.003	0.500	1
Broker	48.184	0.500	0.001	0.499	1
Broker	49.166	0.505	0.004	0.501	1
Broker	50.258	1.951	0.005	1.946	1
Broker	51.300	0.910	0.003	0.907	1
Broker	53.388	1.520	0.005	1.515	1
Broker	52.333	22.310	0.004	22.305	1
Broker	55.392	19.298	0.001	19.297	1
Broker	54.376	20.314	0.003	20.311	1
Broker	56.359	18.333	0.003	18.329	1
Broker	56.943	20.165	17.651	2.514	1
Ensemble aggregation	01:05.050	12.071	9.545	2.525	1
Broker	01:17.128	1.619	0.001	1.618	1

Table 5.3: Detailed comparison of execution times of all tasks run in the example workflow using eight nodes as well as the node the task ran on.

List of Figures

Listings

References

[1] NASA's Jet Propulsion Laboratory. *Global Land-Ocean temperature index*. July 2020. URL: climate.nasa.gov/vital-signs/global-temperature.

[2] M. Roser and H. Ritchie. *Technological Progress*. July 2020. URL: ourworldindata.org/technological-progress.

[3] E. Strohmaier et al. *Performance Development*. July 2020. URL: www.top500.org/statistics/perfdevel.

[4] Martin Schultz. *Pilot Lab Exascale Earth System Modelling*. Oct. 2020. URL: fz-juelich.de/SharedDocs/Meldungen/IAS/JSC/EN/2019/2019-09-pl-exaesm.html?nn=2430110.

[5] The Trustees of Princeton University. *What is an Earth System Model (ESM)?* July 2020. URL: soccom.princeton.edu/content/what-earth-system-model-esm.

[6] *Earth System Science: Overview: A Program for Global Change*. Washington, DC: The National Academies Press, 1986. DOI: 10.17226/19210. URL: https://www.nap.edu/catalog/19210/earth-system-science-overview-a-program-for-global-change.

[7] University corporation for atmospheric research. *Community Earth System Model*. July 2020. URL: www.cesm.ucar.edu.

[8] Earth System Model Computational Infrastructure. *Common Infrastructure for Modeling the Earth*. July 2020. URL: github.com/ESMCI/cime.

[9] Polarising. *How a Hub-and-Spoke architecture can help manage data*. July 2020. URL: www.polarising.com/2018/09/hub-spoke-architecture-can-help-manage-data.

[10] ExtremeTech. *The history of supercomputers*. Oct. 2020. URL: www.extremetech.com/extreme/125271-the-history-of-supercomputers.

[11] Gordon E. Moore. *Cramming more components onto integrated circuits*. Oct. 2020. URL: newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf.

[12] Steve Blank. *What the GlobalFoundries' Retreat Really Means*. Oct. 2020. URL: spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means.

[13] Forschungszentrum Jülich. *JUWELS - Configuration*. Aug. 2020. URL: `fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/Configuration/Configuration_node.html`.

[14] ThinkParQ. *BeeGFS the Parallel Cluster File System*. July 2020. URL: `www.beegfs.io/content`.

[15] SchedMD. *Slurm workload manager*. July 2020. URL: `slurm.schedmd.com/slurm.conf.html`.

[16] C. Evans. *YAML Ain't Markup Language*. July 2020. URL: `yaml.org`.

[17] The Linux Foundation. *gRPC a high-performance, open source universal RPC framework*. July 2020. URL: `grpc.io`.

[18] N. Naaman and R. Rom. "Packet scheduling with fragmentation". In: *Proceedings: Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 1. 2002, 427–436 vol.1.

[19] S. K. Korwar, S. Vadhiyar, and R. S. Nanjundiah. "GPU-enabled efficient executions of radiation calculations in climate modeling". In: *20th Annual International Conference on High Performance Computing*. 2013, pp. 353–361.