Ergonomics and efficiency of workflows on HPC clusters

Doctoral Thesis Proposal

Jakub Beránek

IT4Innovations National Supercomputing Center
VSB – Technical University of Ostrava
Czech Republic

August 2022

Supervisor
Ing. Jan Martinovič, Ph.D.

Contents

1 Introduction

High Performance Computing (HPC) infrastructures are crucial for the advancement of scientific research, as they offer unparalleled computing resources that can be leveraged to perform the most complex scientific experiments. Over the last several decades, the performance of HPC clusters has been increasing steadily, effectively doubling every few years. However, such gains are not always for free. One notable cost associated with the growth of HPC performance is its increasing complexity. Computing nodes now consist of hundreds of cores, making it quite challenging to write programs that can scale to such high core counts. They contain hundreds of gigabytes of memory, which necessitates non-uniform memory architectures (NUMA) that require specialized programming techniques to achieve optimal performance. HPC clusters are also becoming increasingly heterogeneous, with devices such as graphics accelerators (GPUs) or reconfigurable hardware (FPGAs) providing most of their (theoretically attainable) performance. Reaching the full potential of these accelerators requires using further specialized programming languages and interfaces.

Historically, optimized HPC software was usually written using system or scientifically focused programming languages (e.g. `C`, `C++` or `Fortran`) and specialized libraries for parallelizing and distributing computation (such as OpenMP and MPI) [19]. While these rather low-level technologies are able to provide the best possible performance, it can be quite challenging and slow to develop (and maintain) applications that use them effectively. And with the advent of heterogeneous clusters, even more technologies (such as `CUDA` for GPUs or `HLS` for FPGAs) have to be adopted. It is unreasonable to expect that most scientists that use HPC resources (who are often not primarily software developers) will be able to use these low-level technologies in an efficient manner without making the development process too slow and cumbersome.

Increasingly powerful hardware enables more complex computations, which are in turn more demanding to develop. Areas such as weather prediction, machine learning model training or big data analysis require launching thousands or even millions of simulations and experiments and require a lot of prototyping during development. These experiments can be quite complex, consisting of multiple dependent steps, such as data ingestion, preprocessing, computation, postprocessing, visualisation etc. It is thus imperative for scientists to have a quick way of prototyping these applications, otherwise the development would just be too slow.

That is why in recent years, HPC users are increasingly moving towards task-based programming paradigms. These paradigms allow users to focus on the problem domain, quickly prototype and describe computation that consists of a large number of individual steps. Using a task-based approach, computations are described using a set of atomic computing blocks (*tasks*) that are composed together in a *task graph* which captures the dependencies between the individual tasks. Using task graphs abstracts away most of the complexity of network communication and parallelization and allows the user to express their computation in a simple way. Combined with the fact that task-based tools often provide interfaces written in high-level languages, such as Python, or various domain-specific languages (DSLs), it makes them an ideal tool for rapid scientific prototyping.

While task graphs are already commonly being used and deployed on various distributed systems, there are certain challenges that limit their development productivity and scalability when deployed on HPC systems specifically. They stem from the interaction with HPC job managers, various quirks commonly present in HPC clusters or from the sheer computational scale that is possible thanks to vast resources provided by HPC systems. Removing or alleviating some of those challenges could lower the barrier of entry and make task graph execution better suited for various HPC use-cases.

The proposed thesis aims to improve task graph execution in HPC by focusing on two important areas, namely developer ergonomics and efficient hardware utilization. The primary goal is to enable computing task graphs on HPC systems in a way that is simple and comfortable for the user, supports essential HPC-specific use-cases and performs in an efficient manner. To achieve this goal, the thesis has the following objectives:

- Identify challenges of task graph execution on HPC systems.
- Design approaches for enabling task graph execution on HPC systems in a way that is both ergonomic for the user and also resource-efficient.
- Implement and integrate the designed approaches in a task graph execution tool to validate them in real world usage.

The thesis proposal is structured as follows. Section 2 introduces task graphs. Section 3 describes current challenges of task graph execution on HPC systems. Section 4 mentions existing task runtime systems and related works. Section 5 enumerates what has been achieved so far. Section 6 summarizes the thesis proposal and outlines future work.

2 Term definitions

This section contains definitions of elementary terms that will be used throughout this proposal. The terminology surrounding tasks and task graphs is quite diverse and both theoretical works and various software tools tend to use slightly differing terms for similar structures. For the sake of simplicity, the description of tasks in this section will be focused on how do the tasks concepts map to their implementation in existing tools for executing task workflows. Some used definitions might thus deviate from terminology used in previous works.

2.1 Task

A *task* is a description of a computation that can be executed in the future. The *execution* of a task is the process of performing the described computation on some specific input value(s) while producing some output value(s). The purpose of tasks is to represent computations in a way that allows treating computations as data. This has many advantages; tasks can be serialized, sent between different nodes in a cluster or stored to disk, and it is possible to execute them an arbitrary number of times.

The task's computation description should contain a specific method of procuring inputs for the computation (if there are any) and a set of outputs that will be produced when the task is executed. Each input can either be specified as a concrete value or the task can describe how should the input be procured when the task is executed. Often, an output of a task is used as an input for another task.

In practice, a task usually corresponds to one of the following two descriptions:

Executing a function A common description of a task is the specification of a function (i.e. a block of code that is part of some program), along with specific inputs that will be passed as arguments for that function. The output of the task is then the value returned by the function when it is executed (called).

It is crucial to note that in this case, a task corresponds to the combination of a function and some specific way of producing its arguments[1], not just to the function itself. Two different tasks might represent the execution of the same function, but with different (ways of procuring) inputs.

[1]In programming language terms, a task corresponds to a lazy invocation of a function, with its parameters bound to specific input expressions.

Executing a program A more coarse-grained way of describing computations is to specify a binary file representing a program that should be executed, along with its specific input values. In this case, the inputs might be e.g. command-line parameters or environment values. The output of a task that executes a program can be e.g. the contents of its standard output or a set of files written to disk.

Tasks can be executed on a *worker*, an abstract computational provider. Distributed task runtimes map workers to hardware resources in various ways. For example, there can be a single worker per computational node or a worker per each core. Usually a single task is executed on a single worker, but in HPC scenarios, there are use-cases for tasks that span multiple workers (this will be further discussed in Section 3).

Tasks can also define various constraints and configuration that influences their execution, e.g. they can specify what kinds of workers are capable of executing them. The term *resource requirements* will be used for such constraints in this proposal. As an example, a task that describes the training of a machine learning model might require a GPU (Graphics Processing Unit) to be present on a worker that wants to compute the task. Other resources might include e.g. a minimum required amount of memory (RAM) or a required amount of processor cores necessary to execute the task. Each resource requirement can be seen as a special case of task input, althought requirements and task inputs are usually treated separately in existing task runtimes.

2.2 Task graph

To build a complex workflow out of individual tasks, we need the ability to compose them together, and also, crucially, to introduce the notion of *dependencies* between tasks, which allow us to define workflows consisting of multiple steps that pass data amongst themselves. A natural way of composing task graphs and expressing dependencies is to build a directed acyclic graph (DAG) of tasks, which we will label as a *task graph*.

In its most basic form, task graph vertices represent tasks and task graph edges define dependencies between tasks. Numerous other things can also be encoded in task graphs, for example it is quite natural to consider the task graph edges to be abstract channels through which the outputs of one task become the inputs of another task that depends on it.

The exact semantics of vertices and edges of task graphs depend heavily on the specifics of tools that implement them. What follows is a formal definition of a simple form of a task graph which captures the basic structure of dependencies between tasks, to provide basic intuition.

Formally, a task graph is a pair (V, E), where V is a set of tasks and $E \subseteq \{(x, y) \mid (x, y) \in V \times V \land x \neq y\}$ is a set of dependencies between tasks. When there exists a dependency (x, y), task y cannot be executed before x has finished executing.

2.3 Task runtime

We will use the term *task runtime* for a system that actually executes computational workflows defined using task graphs. There are many existing task runtimes, with varying architectures, features and trade-offs, which affect factors like performance, fault tolerance or developer productivity. Some of these existing runtimes will be described in more detail in Section 4.

In general, a task runtime has to manage and monitor all aspects of task graph execution. One of these aspects is managing the state of computational resources (workers) that actually execute tasks. For example, it has to handle the lifetime of workers (their connection/disconnection), facilitate data transfers between them or provide resiliency in case of worker failures. Another aspect is the management of tasks themselves. It has to keep track of which tasks have already been executed or have failed, which tasks are currently executing on some worker(s) and which tasks can be executed next because their task dependencies have already been resolved. Worker and task state management is especially complex in a distributed setting, where the workers operate on remote computational nodes connected by a network.

The runtime is also responsible for actually executing the tasks on workers. From the perspective of the task runtime, a task is an atomic element that cannot be further divided; it is simply an opaque structure that can be executed.

2.4 Task scheduling

Another important aspect that needs to be implemented by a task runtime is *task scheduling*, which is described in its own section to emphasize that it is a crucial part of task graph execution.

One of the main benefits of programming paradigms that leverage task graphs is that they are *implicitly parallel*. With e.g. MPI, the user has to explicitly state which nodes should communicate together, how should the data be serialized and what communication patterns should be used. Using task graphs, the user simply describes what should be computed (tasks) and how is the computation logically ordered (task dependencies).

The goal of a task runtime is to analyse the available parallelism contained within a task graph and plan (*schedule*) the execution of tasks on specific workers in a way that optimizes some key metric. There are multiple metrics being used, such as the latency to execute specific critical tasks, but the most common metric is *makespan* – the duration between the start of the execution of the first task to the completion of all tasks within the task graph.

The problem of optimal scheduling of tasks onto workers is NP-hard [28], even in the most basic scenarios (e.g. even if the exact duration of executing each task is known, and even if we do not consider network costs of transferring data between workers). Task runtimes thus have to resort to various heuristics tailored to their users' needs. Some classic task scheduling heuristics and their comparisons can be found in [9, 18, 15, 29, 1].

The scheduling heuristics of the runtime have to take many factors into consideration when deciding on which worker should a task execute:

Resource requirements If a task specifies any resource requirements, they have to be respected by the scheduler, therefore the runtime must observe the (dynamically changing) available resources of each worker and schedule tasks accordingly to uphold their requirements.

Data transfer cost If the runtime operates in a distributed cluster, one of the most important scheduling aspects that it needs to consider is the transfer cost of data between workers over the network. All benefits gained by computing a task on another worker to achieve parallelization might be lost if it takes too much time to send the data (task outputs) to that worker.

The scheduler thus has to carefully balance the communication-to-computation ratio, based on the available network bandwidth, sizes of outputs produced by tasks and the current utilization of workers.

Scheduling overhead The overhead of computing the scheduling decisions itself also cannot be underestimated. As already stated, computing an optimal solution is infeasible, but even heuristical approaches can have wildly different performance characteristics. Producing a lower quality schedule sooner, rather than a higher quality schedule later, can be sometimes beneficial, as we have demonstrated in [2].

3 Task graphs in HPC

Even though task graphs are commonly being executed on distributed systems, executing them on HPC clusters specifically presents unique challenges, both in terms of efficient and scalable task execution, and of programmer productivity and ergonomics when designing and prototyping HPC-scale task graphs.

This section discusses several challenges of executing task graphs on HPC systems and also use-cases that are fairly unique in the HPC environment. In order to enable truly ergonomic and efficient execution of task graphs, task runtimes should support the mentioned use-cases and should also be aware of the challenges, and ideally deal with them as best they could. This section will thus also mention various properties and features that task runtimes should offer to deal with the mentioned challenges.

3.1 Job manager

The vast majority of HPC systems use some kind of job manager (often PBS/Torque [25] or Slurm [30]) to facilitate job submission, resource management and project accounting [24]. To perform any computation using a job manager, the user has to submit a job that describes how many nodes they want to allocate and what is the expected runtime of their computation. The job is then submitted into the job manager *queue* and starts to execute only once there are enough free computational resources. Job managers are used to provide fair access to HPC resources that avoids oversubscription and also to handle accounting of computation. They tend to have fairly strict limits on the number of jobs that users can submit and the amount of nodes that they can have reserved for their jobs at any given time.

To distinguish the often overloaded terms *task* and *job*, we will use the following definitions in the rest of the text. The term *task* will be used for description of a computation that can be very fine-grained (e.g. a function call or an execution of a single program), is composed with other tasks in a task graph and executed by a task runtime. The term *job* will be reserved for coarse-grained HPC jobs submitted to job managers (like Slurm or PBS/Torque), which tend to run for hours or even days and often span multiple computational nodes.

In a way, HPC job managers can also be viewed as task runtimes that operate on a very coarse level – their tasks are HPC jobs that can span hundreds of nodes, run for days or even longer and consist of many various programs being executed. In theory, users could submit fine-grained task graphs to these job managers, which could serve as a natural way for computing complex workflows on HPC systems and thus facilitate their usage.

In practice, it is not always feasible to use the current popular job managers (Slurm and PBS/Torque) in this way, because they operate on a level that is far too coarse-grained for large and complex task graphs. Their overhead for scheduling and executing a single job is orders of magnitude larger than for typical task runtimes (seconds vs milliseconds) and their support for dependencies between tasks/jobs is very basic.

Another problem is node granularity. For small tasks that use only a few cores, we would like to schedule multiple tasks onto a single node at the same time. While job managers are able to create jobs that require only a fraction of a node, this functionality is disabled on some clusters[2], to avoid processes of multiple users running on the same node at the same time, and also because the overhead of executing that many jobs could be unmanageable.

This creates a certain dichotomy between the coarse-grained job manager and (usually a fine-grained) task runtime, and instead of facilitating simple usage of HPC clusters, it creates a barrier for users. Instead of building a task graph of the whole computation and executing it with a single command, they have to think about how to map the task graph to HPC jobs in order to be able to execute their tasks on an HPC cluster and also to amortize the overheads of the job manager.

It should be noted that even though there is definitely room for improving the performance of HPC job managers, some of their complexity and performance limitations are inherent. They have to provide accurate accounting, handle robust and secure cleanup of jobs, take care of job and process isolation, ensure user fairness and many other things. Many of these responsibilities are out of scope for task runtimes, which enables them to achieve higher performance.

There are several approaches that can be used to map task graphs to HPC jobs. Here are some examples of these approaches, ordered from the simplest to the most complicated:

Execute the whole task graph in a single job If the task graph does not have a large number of tasks, or it can be executed quickly, it could be submitted in a single job. This approach would mostly be as simple for the user as executing the task graph on a non-HPC cluster or a personal computer. However, since jobs are bound both by strict

[2]`https://docs.it4i.cz/general/resource_allocation_and_job_execution/#job-submission-and-execution`

node count and time limits, this approach will only be usable for rather small task graphs.

Indeed, if the computation is short, it probably does not even make sense to even use an HPC cluster to compute it. A more realistic scenario is that the user has a large number of small task graphs to execute, but this situation can be seen as a special case of a large task graph that consists of many disjoint components (subgraphs).

Execute each task as an individual job While this is certainly tempting, since this approach is mostly straightforward to implement using existing job managers, it is impractical because of the mentioned difference in granularity between tasks and jobs. Job managers have an enormous overhead per each job, and furthermore they seldom allow the user to create more than a few hundreds of jobs at the same time, both to provide fairness and also because they simply cannot scale to such amount of jobs. Furthermore, a single job often has to span at least a (complete) single node, and if a single task cannot leverage a whole computational node, this would lead to wasting resource.

Split the task graph into a smaller amount of jobs This is the ultimate approach that the user has to resort to if their task runtime does not provide any special support for job managers. The task graph has to be split into smaller parts which will then be executed in individual jobs. In addition to manually splitting the task graph, additional infrastructure has to be implemented, for example to store the intermediate results of the computed tasks before the jobs ends, to merge the intermediate results from multiple jobs and also to periodically submit new jobs until the whole task graph is computed.

This reduces the ergonomics of using task graphs, because it basically forces the user to reimplement part of the task runtime behavior to overcome the limitations of job managers. Splitting the task graph into a fixed amount of jobs also has the disadvantage that the individual tasks cannot be load balanced across jobs, even if multiple jobs run concurrently, because each job will simply execute its own separate copy of some task runtime that will execute a part of the task graph.

This dichotomy creates a large gap for users attempting to scale their task graph computation. Running on a personal computer tends to be quite simple. After that, moving to an HPC cluster and executing the entire task graph inside a single job is also quite straightforward. But once the task graph has to be split into multiple jobs, the nice abstraction of implicitly parallel task graphs that can be executed with a single command quickly falls apart, as the user has to perform a lot of additional work to make this scenario work efficiently.

Ideally, the users would not have to deal with the job manager; they should be able to construct a task graph and execute it directly on an HPC cluster in a straightforward way. This could be achieved either by adding support for executing fine-grained task graphs to job managers or by integrating task runtimes with job managers to provide transparent execution of task graphs on HPC systems.

3.2 Cluster heterogeneity

In recent years, HPC clusters have started to become increasingly heterogeneous. This trend can be clearly seen in the TOP500 list of most powerful supercomputers [16]. Individual cluster nodes contain varying amounts of cores, memory, NUMA nodes or accelerators like GPUs or FPGAs. This complexity also propagates to task definitions and their requirements. Some tasks can be single-threaded, some multithreaded, some are offloadable to accelerators if there is one available, some can only run on accelerators, while some of them can only execute on CPUs.

To uphold these task requirements, it should be possible for users to define fine-grained task resource requests (e.g. "this task requires two GPUs, sixteen cores and at least 32 GiB of memory"). To match these requests, it should be possible to attach arbitrary resources to computational providers (workers). The task runtime should then manage the dynamic resource allocations of workers to individual tasks to make sure that tasks only execute on workers that have enough resources.

A unique resource requirement that is fairly specific to HPC systems is the requirement of using multiple nodes (workers) per a single task. This requirement is necessary for executing programs that are designed to be executed in a distributed fashion, such as programs using MPI, which are quite common in HPC software. This use case is discussed further below.

3.3 Data transfers

After a task is computed, it can produce various data outputs, standard error or output streams, files created on the disk or data objects that are then passed as inputs to dependent tasks. There are many ways of storing and transferring these outputs. Some task frameworks store task outputs on the filesystem, since it is relatively simple to implement, and it provides support for basic data resiliency out-of-the-box.

HPC nodes often do not contain any local disks, instead they tend to use shared filesystems accessed over a network. While this might be seen as an advantage, since with a shared filesystem it is much easier to share task outputs amongst different workers, it can also be a severe bottleneck. Shared, networked filesystems can suffer from quite high latency, and accessing them can consume precious network bandwidth that is also used e.g. for managing computation (sending commands to workers) or for direct worker-to-worker data exchange. Furthermore, data produced in HPC computations can be quite large, and thus storing it to a disk can be a bottleneck even without considering networked filesystems.

It should be possible to alleviate these bottlenecks, for example by directly transferring task outputs between workers over the network (preferably without accessing the filesystem in the fast path), by streaming outputs between tasks without the need to store them or by leveraging RAM disks [12]. Making use of HPC specific technologies, such as MPI or InfiniBand, could be also worthwhile to leverage the very fast interconnects available in HPC clusters.

Data outputs produced by tasks tend to be considered immutable, since a single output can be used as an input to multiple tasks, and these might be executed on completely different computational nodes. A problem that can arise with this approach is that if the data outputs are large, but the computation within tasks that work with the data is short, the serialization overhead (or even memory copy overhead, if the dependent task is executed on the same node) starts to dominate the execution time. To support such scenarios, some support for stateful data management can be useful, for example in the form of *actors*, which can be considered stateful tasks that operate on a single copy of some large piece of data.

3.4 Fault tolerance

Some level of fault tolerance should be provided by all task runtimes, but HPC systems have specific requirements in this regard. As was already mentioned, computational resources on HPC clusters are provided through job managers. These node allocations have a temporary duration, which means that for long-running task graphs, workers will disconnect and new workers will connect dynamically during the execution of the task graph. Furthermore, since the job manager allocations go through a queue, it can take some time before new computational resources arrive, therefore the task graph can remain in a paused state where no tasks are being executed, for potentially long periods of time.

Task runtimes should be prepared for these situations; they must handle worker disconnection gracefully, even if that worker was currently executing some task, and they should be able to restart previously interrupted tasks on newly arrived workers. In HPC scenarios, worker instability and frequent disconnects should be considered a common behaviour, not just a rare edge case.

3.5 Multi-node tasks

Many existing HPC applications are designed to be executed on multiple (hundreds or even thousands) nodes in parallel, using MPI libraries or other communication frameworks. To properly support these use-cases, we can introduce multi-node tasks by creating a special resource requirement, which states that a task should be executed on multiple workers at once.

Support for multi-node tasks affects many design areas of a task runtime:

Scheduling When a task requires multiple nodes for execution and not enough nodes are available at a given moment, the scheduler has to decide on a strategy that will allow the multi-node task to execute. If it was constantly trying to backfill available workers with single-node tasks, the multi-node tasks could be starved.

The scheduler might thus have to resort to keep some nodes idle for a while to enable the multi-node task to start as soon as possible. Another approach could be to interrupt the currently executing tasks and checkpoint their state to make space for a multi-node task, and then resume their execution once the multi-node task finishes.

In a way, this decision-making already has to be performed on the level of individual cores even for single-node tasks, but adding multiple nodes per task makes the problem much more difficult.

Data transfers It is relatively straightforward to express data transfers between single-node tasks in a task graph, because they naturally correspond to dependencies (edges) between the tasks. With multi-node tasks, the data distribution patterns become more complex, for example data can be replicated from a single node to multiple nodes when a multi-node task starts or gathered (reduced) from multiple nodes to a single node when such task finishes.

When several multi-node tasks depend on one another, the task runtime should be able to exchange data between them in an efficient manner. This might require some cooperation with the used communication framework (e.g. MPI) to avoid needless repeated serialization and deserialization.

Fault tolerance When a node executing a single-node task crashes or disconnects from the runtime, its task can be rescheduled to a different worker. In the case of multi-node tasks, failure handling requires more communication and is generally more complex. When a task is executing on four nodes and one of them fails, the runtime has to make sure that the other nodes will be notified of this situation, so that they can react accordingly (either by finishing the task with a smaller amount of nodes or by also failing immediately).

To enable common HPC usecases, task runtimes should be able to provide some support for multi-node tasks and allow them to be combined with single-node tasks. Advanced multi-node task support could be provided e.g. by offering some kind of integration with MPI or similar common HPC technologies.

3.6 Scalability

The sheer amount of HPC performance (node count, core count, network interconnect bandwidth) opens up opportunities for executing large scale task graphs, but that in turn presents unique challenges for task runtimes. Below you can find several examples of bottlenecks that might not matter in a small computational scale, but that can become problematic in the context of HPC-scale task graphs.

Task graph materialization Large computations might require building massive task graphs that contain millions of tasks. The task graphs are typically defined and built outside the task runtime itself, for example on the login nodes of computing clusters or on client devices (e.g. laptops), which can provide only relatively low performance. It can be quite slow to build, serialize and transfer such graphs over the network to the task runtime. This can create a bottleneck even before any task is executed. This has been identified as an issue in existing task runtimes [26].

In such case, it can be beneficial to provide an API for defining task graphs in a symbolic way, for example by representing a potentially large group of similar tasks by a single entity. Such symbolic graphs could then be sent to the runtime in a compressed form and re-materialized only at the last possible moment. In an extreme form, the runtime could operate on such graphs in a fully symbolic way, without ever materializing them.

Communication overhead Scaling the number of tasks and workers will necessarily put a lot of pressure on the communication network, both in terms of bandwidth (sending large task outputs between nodes) and latency (sending small management messages between the scheduler and the workers). Using HPC technologies, such as MPI or a lower-level interface like RDMA (Remote Direct Memory Access), could provide a non-trivial performance boost in this regard.

As we have demonstrated in [4, 3], in-network computing, an active area of research, can be also used to optimize various networking applications by offloading some computations to an accelerated NIC (network interface controller). This approach could also be leveraged for task runtimes, for example by reducing the latency of management messages between the scheduler and workers or by increasing the bandwidth of large data exchanges amongst workers, by moving these operations directly onto the network card.

Runtime overhead As we have shown in [2], task runtimes with a centralized scheduler have to make sure that their overhead remains manageable. Even with an overhead of just $1ms$ per task, executing a task graph with a million tasks would result in total accumulated overhead of twenty minutes! Our results indicate that increasing the performance of the central scheduling and management component of a task runtime can have a large positive effect on the overall time it takes to execute the whole task graph.

However, the performance of the central server cannot be increased endlessly, and from some point, using a centralized architecture, which is common to task runtimes, itself becomes a bottleneck. Even if the workers exchange large output data directly between themselves, any single, centralized component may become overloaded simply by coordinating and scheduling the workers.

In that case, a decentralized architecture could be leveraged to avoid the reliance on a central component. Such a decentralized architecture can be found e.g. in Ray [22]. However, to realize the gains of a decentralized architecture, task submission itself has to be decentralized in some way, which might not be a natural fit for common task graph workflows. If all tasks are generated from a single component, the bottleneck will most likely remain even in an otherwise fully decentralized system.

3.7 Iterative computation

A natural way of executing task graphs is to describe the whole computation with a single task graph, submit the graph to the task runtime and wait until all the tasks are completed. However, there are some computations that need a more iterative approach. Training a machine learning model can be stopped early if the loss is no longer decreasing. A chemical or physical simulation is only considered completed once a desired accuracy has been reached, which might take a previously unknown number of steps. These scenarios, and many others like them, are quite common in HPC use cases.

To support iterative computation, task runtimes should allow the user to stop the execution of a task graph (or its subgraph) once a specific condition is met, and also to add new tasks to the task graph in a dynamic fashion, if it is discovered that more iterations are needed.

3.8 Summary

Even though more HPC use-cases and oddities could always be found, it is already clear from the mentioned challenges that HPC use-cases that leverage task graphs can contain a lot of complexity. It could be possible to add support for some mentioned requirements to existing task runtimes, which are described in the next section. However, the described challenges are so diverse and complex that a dedicated approach which considers them holistically could provide a better solution that would avoid both ergonomics and performance from being compromised.

The aforementioned requirements will serve as a basis for further research in the proposed thesis. The goal of the thesis is to design approaches for executing task graphs on HPC systems that take the aforementioned requirements into account. These approaches will leverage the *HyperQueue* task runtime, which is described further in Section 5.4.

4 State of the art

There are many existing task runtime systems, with different usability and performance trade-offs [23, 13, 17, 27, 10, 22]. Even though most of existing task runtimes are quite versatile and can run on pretty much anything from a laptop to a large distributed cluster, they also suffer from various shortcomings in regard to the challenges described in Section 3.

Below you can find a list of notable task runtimes that are representatives of task runtime categories with certain properties (for example centralized/decentralized scheduling, relation to the job manager etc.).

Dask is a task runtime written in Python that is very popular within the Python scientific and data analysis community[23, 11]. It allows users to construct arbitrary task graphs out of Python function calls and then execute them on a distributed cluster. Its runtime architecture is standard, a centralized server receives tasks from clients and then schedules them to a set of connected workers.

As we have analyzed in [2], the fact that *Dask* is written in Python can severely limit its scaling potential for large HPC-scale task graphs. It also does not have support for arbitrary resource requirements. *Dask* itself has no notion of HPC job managers, therefore for large task graphs that span more than a single job, users have to manually split the task graph into multiple jobs and execute a *Dask* cluster computation inside each one of them. There is a helper package called Dask-Jobqueue[3] which can overcome this issue by sharing a single server amongst multiple HPC jobs and thus allowing load balancing of the whole task graph.

Ray is a distributed task runtime that focuses on machine-learning applications and actors (stateful computations useful for iterative processes and model training) [22]. In many aspects, it is quite similar to *Dask*. However, it has a rather unique architectural property – instead of a centralized scheduler, it uses a decentralized scheduling scheme. It leverages Redis as a distributed key-value store, which is uses for both for storing task output data and for storing the state of scheduling itself.

This allows *Ray* to scale very efficiently; in extreme cases, it is able to schedule millions of tasks per second. However, to achieve this performance, the computational workflows have to use a slightly different programming paradigm. They have to be able to create new tasks from workers dynamically, while the workflow is being executed. In that case, the workers can schedule and execute these dynamic tasks locally, or send them to the distributed scheduler store if they are overloaded. If the whole task graph is created on a single client, the bottlenecks that exist in centralized schedulers would also affect Ray's decentralized architecture.

Ray has support for custom resource requirements which it takes into

[3]`https://jobqueue.dask.org/en/latest/`

account while making scheduling decisions, but it does not have built-in support for running outside HPC job managers.

SnakeMake is a workflow management system designed for executing reproducible scientific experiments [17]. Unlike *Dask* and *Ray*, which primarily use imperative Python APIs for building task graphs, *SnakeMake* workflows are specified declaratively in workflow files that leverage a combination of a custom DSL and Python. *SnakeMake* supports custom resources and complex task configuration.

In terms of the job manager, *SnakeMake* can both submit tasks as individual jobs, or it can batch groups of tasks into a single job. However, the grouping is performed statically, and tasks are not load balanced across different jobs.

5 Current progress

This section describes two main bodies of work that have been conducted so far. The first is a set of publications [1, 2, 3, 4] related to the first objective of the thesis, identifying HPC task graph challenges. The second is the work on designing and implementing *HyperQueue*, a task graph execution tool designed for HPC use-cases.

5.1 Task scheduler analysis and benchmarking

The scheduler component, which assigns tasks to individual workers, is one of the most crucial parts of a task runtime, because scheduling decisions can severely affect the duration required to execute the whole task graph. Since task scheduling is NP-hard, various heuristic algorithms are used in practice. These algorithms can suffer from non-obvious edge cases that produce bad quality schedules and also from low runtime efficiency, which can erase any speedup gained from producing a higher quality schedule.

To better understand the behaviour and performance of various scheduling algorithms, we have performed an extensive analysis of several task scheduling algorithms in *Analysis of workflow schedulers in simulated distributed environments* [1]. We have benchmarked several task schedulers under various conditions, including parameters that have not been explored so far, like the minimum delay between invoking the scheduler or the amount of knowledge about task durations available for the scheduler.

Our analysis has shown that despite its simplicity, the foundational HLFET algorithm [9] produces high quality schedules in various scenarios and should thus serve as a good baseline scheduler for task runtimes.

We have also found out that even a completely random scheduler can be competitive with other scheduling approaches for certain task graphs and cluster configurations.

One of the contributions of this work was *Estee*, a simulation framework for task schedulers that is available as an open-source software[4]. It can be used to define a cluster of workers, connect them together using a configurable network model, implement a custom scheduling algorithm and test its performance on arbitrary task graphs. *Estee* also contains implementations of several task scheduler baselines from existing literature and a task graph generator that can be used to generate randomized graphs with similar properties as real-world task graphs. This tool serves as an experimentation testbed for task runtime and scheduler developers.

I have collaborated on this work with Stanislav Böhm and Vojtěch Cima, we have all contributed equally to this work.

5.2 Task runtime bottleneck analysis and optimization

The scheduler is not the only part of a task runtime that can cause bottlenecks in task graph execution. We have analysed an existing and quite popular task runtime *Dask* [23] in *Runtime vs Scheduler: Analyzing Dask's Overheads* [2], both to find out what bottlenecks does it have, and also to benchmark various scheduling algorithms with *Dask*, to test them "in the wild" and thus better validate our results from [1].

Our analysis has demonstrated that *Dask* was bottlenecked not so much by its scheduler, but by the runtime (in)efficiency of its central server. The inefficiencies were caused partly by the design of its communication protocol, but mainly by the fact that *Dask* is implemented in Python, which makes it difficult to fully utilize the available hardware potential. We have also found out that it was impractical to swap *Dask*'s scheduler implementation for another one, since its built-in work-stealing scheduling algorithm was quite firmly integrated into its components.

In order to measure how could *Dask*'s performance be improved if it had a more efficient runtime, as a second main contribution of this work we have developed *RSDS*, an open-source drop-in replacement for the *Dask* central server[5]. It was built from the ground up with a focus on runtime efficiency and scheduler modularity, but at the same time we have designed it to be compatible with the *Dask* protocol, so it could be used by existing *Dask* users to speed up their task workflows.

[4]`https://github.com/it4innovations/estee`

[5]`https://github.com/it4innovations/rsds`

We have performed a series of experiments where we have compared the performance of *RSDS* vs *Dask*. Since *RSDS* allows its users to plug in a different scheduling algorithm easily, we have also compared the performance of *RSDS* with various scheduling algorithms. The experiments were conducted on task graphs generated from tracing real-world *Dask* task workflows, to make sure that we were benchmarking realistic use-cases.

The results of our experiments indicate that optimizing the runtime is definitely a worthy effort to pursuit, as *RSDS* has been able to outperform *Dask* in various scenarios, even though it used a much simpler work-stealing scheduling algorithm. We have also been able to validate our results from [1], for example that even a random scheduler is indeed competitive with other scheduling approaches in many scenarios.

We have contacted the authors and maintainers of *Dask* and discussed our *RSDS* approach with them[6,7,8]. Some of its ideas have been adapted in the *Dask* project and led to improving its performance.

An interesting insight regarding task schedulers that we have gained from our work done in [1, 2] is just how much important are the specific details of scheduling algorithm implementations. While trying to implement various scheduling algorithms from existing literature, we have realized that they are often incomplete. Details like how often should the algorithm be invoked or how to choose between workers which receive an equal scheduling priority from the algorithm are often left up to the implementor. However, our experiments have shown us that these seemingly minor details can have a significant effect on the performance of the scheduler, both in terms of runtime efficiency and the quality of its generated schedules.

I have collaborated on this work with Stanislav Böhm, we have both contributed equally to this work.

5.3 In-network computing

In-network computing is a relatively recent area of research that attempts to explore the possibility of offloading certain computing operations directly to network controllers, in order to massively reduce latency and improve bandwidth of distributed (HPC) applications. The classic approach of performing operations on data sent over the network is to pass it from the network interface through all levels of the CPU memory hierarchy to the processor, which performs the operation, and then send the data all the way back to

[6]`https://github.com/dask/distributed/issues/3139`
[7]`https://github.com/dask/distributed/issues/3783`
[8]`https://github.com/dask/distributed/issues/3872`

the network interface and further into the network. Even though a lot of fundamental networked operations (like collective reductions or key-value operations) are rather simple to compute, they can suffer from high latency because of the data movement needed to get the data to the processing element (CPU).

The idea of in-network computing is to move some data processing operations directly onto a network controller (which can be a NIC connected to a computer, but also e.g. a smart network switch), in order to remove the latency caused by the CPU memory hierarchy. This approach is similar e.g. to offloading expensive computation from the CPU to graphics accelerators (GPUs), even though in the case of network controller offloading, the latency is usually more important than bandwidth (for GPUs it is typically the other way around).

We have explored how offloading computation to a network accelerator could improve the performance of non-contiguous memory transfers in MPI applications [4]. This research was then expanded and generalized to a more general in-network computing framework in [3]. Even though this research area is still fairly unexplored, the results so far look quite promising.

Task runtimes need to exchange many kinds of data amongst workers and the scheduler over the network. Some of these data exchanges could be offloaded to network controllers using either more traditional RDMA (Remote direct memory access) or more general in-network computing methods, in order to reduce the overhead of the task runtime. This could be a good fit for HPC clusters, since network controllers capable of offloading computations are already making their way into this area. One example of such technology is NVIDIA SHARP (Scalable Hierarchical Aggregation and Reduction Protocol), which allows offloading collective operations of MPI programs to networking devices.

I have collaborated on this work with multiple researchers from ETH Zurich. My main contribution was the design, implementation and benchmarking of a virtualized in-network compute engine that was used for implementing non-contiguous data transfers.

5.4 HyperQueue

HyperQueue is an HPC-tailored task runtime designed for executing task graphs in HPC environments. Its two primary objectives are to be as performant as possible and to be easy to use and deploy. It is developed in the

Rust programming language and available as an open-source software[9].

The key idea of *HyperQueue* is to disentangle the submission of computation and the provision of computational resources. With traditional HPC job managers, the computation description is closely tied to the request of computational resources, which leads to problems mentioned in Section 3, such as less efficient load balancing or the need to manually aggregate tasks into jobs. *HyperQueue* separates these two actions; users submit task graphs independently of providing computational resources (workers) and let the task runtime take care of matching them together, based on requested resource requirements and other constraints.

One of the driving use-cases for *HyperQueue* is efficient node usage and load balancing. The latest HPC clusters contain a large number (hundreds) of cores, yet it is quite challenging to design a single program that can scale effectively with so many threads. Thus, in order to fully utilize the whole computational node, multiple tasks that each leverage a smaller amount of threads have to be executed on the same node at once. *HyperQueue* is able to effectively schedule tasks to utilize all available computational nodes, and thanks to its design, it is able to do this not just within a single HPC job, but across many jobs at once.

HyperQueue is being used by users of various HPC centres, and it is also a key component of the Horizon 2020 European Union projects LIGATE[10], EVEREST[11] and ACROSS[12]. It is also envisioned as one of the primary ways of executing computations on the LUMI supercomputer [20].

HyperQueue has been in development for over a year. It should be noted that while I am one of its two primary authors and contributors, *HyperQueue* is a team effort, and it is being developed by multiple people.

6 Future work

The research related to this thesis that has been conducted so far has been mostly focused on analysing task schedulers and task runtime bottlenecks, which are both part of the challenges mentioned in Section 3. In terms of the areas discussed in Section1, these concepts fall into the area of efficiency.

In future work, I plan to also focus on the ergonomics side of HPC task graph execution, primarily on overcoming the issue of integrating task graph

[9]`https://github.com/it4innovations/hyperqueue`

[10]`https://www.ligateproject.eu`

[11]`https://everest-h2020.eu`

[12]`https://across-h2020.eu`

execution with HPC job managers.

HyperQueue is a step in this direction, since it already makes the execution of task graphs on HPC clusters simpler. I have designed and implemented an *automatic allocator* into *HyperQueue* that is able to ask HPC job managers for computational resources automatically, depending on the current computational load. This frees users from interacting with the job manager directly and moves one step closer to a straightforward execution of task graphs on HPC systems. In future work, I plan to improve automatic allocation and better understand its behavior in various HPC use-cases, and also to design new approaches for resolving the mentioned HPC challenges and apply them to real-world HPC scenarios while leveraging *HyperQueue*.

7 Publications

Author's publications related to the thesis

[1] Jakub Beránek, Stanislav Böhm, and Vojtěch Cima. "Analysis of workflow schedulers in simulated distributed environments". In: *The Journal of Supercomputing* 78.13 (Sept. 2022), pp. 15154–15180. ISSN: 1573-0484. DOI: `10.1007/s11227-022-04438-y`.

[2] Stanislav Böhm and Jakub Beránek. "Runtime vs Scheduler: Analyzing Dask's Overheads". In: *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 2020, pp. 1–8. DOI: `10.1109/WORKS51914.2020.00006`.

[3] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beránek, Luca Benini, and Torsten Hoefler. "A RISC-V in-Network Accelerator for Flexible High-Performance Low-Power Packet Processing". In: *Proceedings of the 48th Annual International Symposium on Computer Architecture.* IEEE Press, 2021, pp. 958–971. ISBN: 9781450390866.

[4] Salvatore Di Girolamo, Konstantin Taranov, Andreas Kurth, Michael Schaffner, Timo Schneider, Jakub Beránek, Maciej Besta, Luca Benini, Duncan Roweth, and Torsten Hoefler. "Network-Accelerated Non-Contiguous Memory Transfers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290.

Author's publications unrelated to the thesis

[5] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture.* New York, NY, USA: Association for Computing Machinery, 2021, pp. 282–297. ISBN: 9781450385572.

[6] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. "GraphMine-Suite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra". In: *Proc. VLDB Endow.* 14.11 (July 2021), pp. 1922–1935. ISSN: 2150-8097. DOI: 10.14778/3476249.3476252.

[7] Stanislav Böhm, Jakub Beránek, and Martin Šurkovský. "Haydi: Rapid Prototyping and Combinatorial Objects". In: *Foundations of Information and Knowledge Systems.* Ed. by Flavio Ferrarotti and Stefan Woltran. Cham: Springer International Publishing, 2018, pp. 133–149. ISBN: 978-3-319-90050-6.

[8] Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, and Torsten Hoefler. "Streaming Message Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290.

References

[9] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. "A Comparison of List Schedules for Parallel Processing Systems". In: *Commun. ACM* 17.12 (Dec. 1974), pp. 685–690. ISSN: 0001-0782. DOI: 10.1145/361604.361619.

[10] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. "Parsl: Pervasive Parallel Programming in Python". In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing.* HPDC '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 25–36. ISBN: 9781450366700. DOI: `10.1145/3307681.3325400`.

[11] Genevieve Buckley. *2021 Dask User Survey.* URL: `https://blog.dask.org/2021/09/15/user-survey` (visited on 06/09/2022).

[12] Vojtěch Cima, Stanislav Böhm, Jan Martinovič, Jiří Dvorský, Kateřina Janurová, Tom Vander Aa, Thomas J. Ashby, and Vladimir Chupakhin. "HyperLoom: A Platform for Defining and Executing Scientific Pipelines in Distributed Environments". In: PARMA-DITAM '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–6. ISBN: 9781450364447. DOI: `10.1145/3183767.3183768`. URL: `https://doi.org/10.1145/3183767.3183768`.

[13] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Barja, Emilio Palumbo, and Cedric Notredame. "Nextflow enables reproducible computational workflows". In: *Nature Biotechnology* 35 (Apr. 2017), pp. 316–319. DOI: `10.1038/nbt.3820`.

[14] Martin Golasowski, Jakub Beránek, Martin Šurkovský, Lukáš Rapant, Daniela Szturcová, Jan Martinovič, and Kateřina Slaninová. "Alternative Paths Reordering Using Probabilistic Time-Dependent Routing". en. In: *Advances in Networked-based Information Systems.* Ed. by Leonard Barolli, Hiroaki Nishino, Tomoya Enokido, and Makoto Takizawa. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2020, pp. 235–246. ISBN: 978-3-030-29029-0. DOI: `10.1007/978-3-030-29029-0_22`.

[15] Tarek Hagras and J Janeček. "Static vs. dynamic list-scheduling performance comparison". In: *Acta Polytechnica* 43.6 (2003).

[16] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, Ali R. Butt, and Youngjae Kim. "An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers". In: *The International Conference on High Performance Computing in Asia-Pacific Region.* HPC Asia 2021. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 11–22. ISBN: 9781450388429. DOI: `10.1145/`

3432261.3432263. URL: https://doi.org/10.1145/3432261.3432263.

[17] Johannes Köster and Sven Rahmann. “Snakemake—a scalable bioinformatics workflow engine”. In: *Bioinformatics* 28.19 (Aug. 2012), pp. 2520–2522. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts480. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/19/2520/819790/bts480.pdf.

[18] Yu-Kwong Kwok and Ishfaq Ahmad. “Benchmarking the task graph scheduling algorithms”. In: *ipps*. IEEE. 1998, p. 0531.

[19] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. “A Large-Scale Study of MPI Usage in Open-Source HPC Applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356176.

[20] *Lumi, Europe’s most powerful supercomputer, is solving global challenges and promoting a green transformation*. June 2022. URL: https://www.it4i.cz/en/about/infoservice/press-releases/lumi-europes-most-powerful-supercomputer-is-solving-global-challenges-and-promoting-a-green-transformation (visited on 07/23/2022).

[21] Jan Martinovič, Martin Golasowski, Kateřina Slaninová, Jakub Beránek, Martin Šurkovský, Lukáš Rapant, Daniela Szturcová, and Radim Cmar. “A Distributed Environment for Traffic Navigation Systems”. en. In: *Complex, Intelligent, and Software Intensive Systems*. Ed. by Leonard Barolli, Farookh Khadeer Hussain, and Makoto Ikeda. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2020, pp. 294–304. ISBN: 978-3-030-22354-0. DOI: 10.1007/978-3-030-22354-0_27.

[22] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. “Ray: A Distributed Framework for Emerging AI Applications”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 9781931971478.

[23] Matthew Rocklin. “Dask: Parallel Computation with Blocked algorithms and Task Scheduling”. In: Jan. 2015, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013.

[24] *SchedMD Slurm usage statistics.* URL: https://schedmd.com/ (visited on 05/18/2022).

[25] Garrick Staples. “TORQUE Resource Manager”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing.* SC ’06. Tampa, Florida: Association for Computing Machinery, 2006, 8–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188464.

[26] *Task graph building performance issue in Dask.* URL: https://github.com/dask/distributed/issues/3783 (visited on 05/25/2022).

[27] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. “PyCOMPSs: Parallel computational workflows in Python”. In: *International Journal of High Performance Computing Applications* 31 (Aug. 2015). DOI: 10.1177/1094342015594678.

[28] J. D. Ullman. “NP-complete Scheduling Problems”. In: *J. Comput. Syst. Sci.* 10.3 (June 1975), pp. 384–393. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80008-0.

[29] Huijun Wang and Oliver Sinnen. “List-Scheduling vs. Cluster-Scheduling”. In: *IEEE Transactions on Parallel and Distributed Systems* (2018).

[30] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing.* Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4. DOI: 10.1007/10968987_3.